Oracle® AI Database

Database Development Guide

26ai
G43974-01
October 2025

Oracle AI Database Database Development Guide, 26ai

G43974-01

Copyright © 1999, 2025, Oracle and/or its affiliates.

Primary Author: Jiji Thomas

Contributing Authors: J. Sharma, A. Kumar, C. Murray, T. Kyte, D. Adams, L. Ashdown, S. Moore, E. Paapanen, R. Strohm, R. Ward

Contributors: D. Alpern, G. Arora, T. Chang, B. Cheng, R. Day, R. Decker, G. Doherty, 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, 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, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. 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®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside 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, Epyc, and the AMD 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 about 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 unless otherwise set forth in an applicable agreement between you and Oracle. 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, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience i
Related Documents i
Conventions ii

Changes in This Release for Oracle AI Database Development Guide

New Features in 26ai i
Deprecated Features iv

Part I Database Development Fundamentals

1 Design Basics

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

1.9.4 Temporal Validity Support 11

2 Connection Strategies for Database Applications

2.1 Design Guidelines for Connection Pools 1
2.1.1 Connection Storms 1
2.1.2 Guideline for Preventing Connection Storms: Use Static Pools 2
2.2 Design Guideline for Login Strategy 3
2.3 Design Guideline for Preventing Programmatic Session Leaks 3
2.3.1 Drained Connection Pools 4
2.3.2 Checking for Session Leaks 4
2.3.3 Lock Leaks 4
2.3.4 Logical Corruption 4
2.4 Using Runtime Connection Load Balancing 5
2.4.1 About Runtime Connection Load Balancing 5
2.4.2 Enabling and Disabling Runtime Connection Load Balancing 6
2.4.2.1 OCI 6
2.4.2.2 OCCI 7
2.4.2.3 JDBC 7
2.4.2.4 ODP.NET 7
2.4.3 Receiving Load Balancing Advisory FAN Events 8

3 Performance and Scalability

3.1 Performance Strategies 1
3.1.1 Designing Your Data Model to Perform Well 1
3.1.1.1 Analyze the Data Requirements of the Application 2
3.1.1.2 Create the Database Design for the Application 2
3.1.1.3 Implement the Database Application 3
3.1.1.4 Maintain the Database and Database Application 4
3.1.2 Setting Performance Goals (Metrics) 4
3.1.3 Benchmarking Your Application 4
3.2 Tools for Performance 5
3.2.1 DBMS_APPLICATION_INFO Package 5
3.2.2 SQL Trace Facility (SQL_TRACE) 6
3.2.3 EXPLAIN PLAN Statement 7
3.3 Monitoring Database Performance 8
3.3.1 Automatic Database Diagnostic Monitor (ADDM) 8
3.3.2 Monitoring Real-Time Database Performance 9
3.3.3 Responding to Performance-Related Alerts 9
3.3.4 SQL Advisors and Memory Advisors 9
3.4 Testing for Performance 10

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.5 Using Client Result Cache 11
3.5.1 About Client Result Cache 11
3.5.2 Benefits of Client Result Cache 12
3.5.3 Guidelines for Using Client Result Cache 13
3.5.3.1 SQL Hints 14
3.5.3.2 Table Annotation 15
3.5.3.3 Session Parameter 15
3.5.3.4 Effective Table Result Cache Mode 16
3.5.3.5 Displaying Effective Table Result Cache Mode 16
3.5.3.6 Result Cache Mode Use Cases 17
3.5.3.7 Queries Never Result Cached in Client Result Cache 17
3.5.4 Setting Result Cache Integrity 17
3.5.5 Client Result Cache Consistency 18
3.5.6 Deployment-Time Settings for Client Result Cache 19
3.5.6.1 Server Initialization Parameters 20
3.5.6.2 Client Configuration Parameters 21
3.5.7 Client Result Cache Statistics 21
3.5.8 Validation of Client Result Cache 22
3.5.8.1 Measure Execution Times 22
3.5.8.2 Query V$MYSTAT 22
3.5.8.3 Query V$SQLAREA 23
3.5.9 Client Result Cache and Server Result Cache 23
3.5.10 Client Result Cache Demo Files 24
3.5.11 Other Supported Features 24
3.5.12 Client Result Cache Compatibility with Previous Releases 25
3.6 Statement Caching 25
3.7 OCI Client Statement Cache Auto-Tuning 26
3.8 Client-Side Deployment Parameters 26
3.9 Using Query Change Notification 27
3.10 Using Database Resident Connection Pool 27
3.10.1 About Database Resident Connection Pool 28
3.10.2 Configuring DRCP 29
3.10.2.1 Managing Permissions for Per-PDB DRCP 30
3.10.3 Using Multi-pool DRCP 31
3.10.3.1 Adding a DRCP Pool 32
3.10.3.2 Removing a DRCP Pool 33
3.10.3.3 About Authentication Pool in Multi-pool DRCP 33
3.10.3.4 Managing the Connection Broker in Multi-pool DRCP 33
3.10.4 Sharing Proxy Sessions 34
3.10.5 Using JDBC with DRCP 34
3.10.6 Using OCI Session Pool APIs with DRCP 35
3.10.7 Session Purity 35

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.10.8 Connection Class 36
3.10.8.1 Example: Setting the Connection Class as HRMS 36
3.10.9 Session Purity and Connection Class Defaults 37
3.10.10 Setting the Purity and Connection Class in the Connection String 37
3.10.11 Starting DRCP 37
3.10.12 Shut Down Connection Draining for DRCP 38
3.10.13 Enabling DRCP 38
3.10.14 Connecting to a Pool in Multi-pool DRCP 39
3.10.15 Implicit Connection Pooling 39
3.10.15.1 Implicit Stateful and Stateless Sessions 41
3.10.15.2 Statement and Transaction Boundary 41
3.10.15.3 Configuring Implicit Connection Pool Boundaries 42
3.10.15.4 Impact of Round-trip OCI Calls on Implicit Connection Pooling States 42
3.10.15.5 Deciding which Pool Boundary to Use 43
3.10.15.6 Implicit Connection Pooling with CMAN-TDM and PRCP 43
3.10.15.7 Setting or Resetting the Session State at the Boundaries During Deployment 44
3.10.15.8 Using the Session Cached Cursors with Implicit Connection Pooling 45
3.10.15.9 Security 45
3.10.16 Benefiting from the Scalability of DRCP in an OCI Application 45
3.10.17 Benefiting from the Scalability of DRCP in a Java Application 46
3.10.18 Best Practices for Using DRCP 46
3.10.19 Compatibility and Migration 48
3.10.20 Using DRCP with Oracle Database Native Network Encryption 48
3.10.21 DRCP Restrictions 49
3.10.22 Using DRCP with Custom Pools 50
3.10.23 Explicitly Marking Sessions Stateful or Stateless 50
3.10.24 Using DRCP with Oracle Real Application Clusters 51
3.10.25 DRCP with Data Guard 51
3.11 Memoptimize Pool 52
3.12 Oracle RAC Sharding 52

4 Designing Applications for Oracle Real-World Performance

4.1 Using Bind Variables 1
4.2 Using Instrumentation 2
4.3 Using Set-Based Processing 2
4.3.1 Iterative Data Processing 2
4.3.1.1 About Iterative Data Processing 3
4.3.1.2 Iterative Data Processing: Row-By-Row 3
4.3.1.3 Iterative Data Processing: Arrays 4
4.3.1.4 Iterative Data Processing: Manual Parallelism 5

Copyright © 1999, 2025, Oracle and/or its affiliates.

4.3.2 Set-Based Processing 8

5 Security

5.1 Enabling User Access with Grants, Roles, and Least Privilege 1
5.2 Automating Database Logins 2
5.3 Controlling User Access with Fine-Grained Access Control 2
5.4 Using Invoker's and Definer's Rights for Procedures and Functions 4
5.4.1 What Are Invoker's Rights and Definer's Rights? 4
5.4.2 Protecting Users Who Run Invoker's Rights Procedures and Functions 5
5.4.3 How Default Rights Are Handled for Java Stored Procedures 5
5.5 Managing External Procedures for Your Applications 6
5.6 Auditing User Activity 6

6 High Availability

6.1 Transparent Application Failover (TAF) 1
6.1.1 About Transparent Application Failover 1
6.1.2 Configuring Transparent Application Failover 2
6.1.3 Using Transparent Application Failover Callbacks 2
6.2 Oracle Connection Manager in Traffic Director Mode 3
6.3 About Fast Application Notification (FAN) 3
6.3.1 About Receiving FAN Event Notifications 4
6.4 About Fast Connection Failover (FCF) 5
6.5 About Application Continuity 6
6.5.1 Reset Database Session State to Prevent Application State Leaks, Use RESET_STATE 7
6.6 About Transaction Guard 8
6.7 About Service and Load Management for Database Clouds 9

7 Advanced PL/SQL Features

7.1 PL/SQL Data Types 1
7.2 Dynamic SQL 1
7.3 PL/SQL Optimize Level 2
7.4 Compiling PL/SQL Units for Native Execution 2
7.5 Exception Handling 2
7.6 Conditional Compilation 2
7.7 Bulk Binding 3

Part II SQL for Application Developers

Copyright © 1999, 2025, Oracle and/or its affiliates.

8 SQL Processing for Application Developers

8.1 Description of SQL Statement Processing 1
8.1.1 Stages of SQL Statement Processing 2
8.1.2 Shared SQL Areas 3
8.2 Grouping Operations into Transactions 4
8.2.1 Deciding How to Group Operations in Transactions 4
8.2.2 Improving Transaction Performance 5
8.2.3 Managing Commit Redo Action 6
8.2.4 Determining Transaction Outcome After a Recoverable Outage 7
8.2.4.1 Understanding Transaction Guard 8
8.2.4.2 Understanding DBMS_APP_CONT.GET_LTXID_OUTCOME 9
8.2.4.3 Using Transaction Guard 12
8.3 Ensuring Repeatable Reads with Read-Only Transactions 12
8.4 Locking Tables Explicitly 13
8.4.1 Privileges Required to Acquire Table Locks 14
8.4.2 Choosing a Locking Strategy 14
8.4.2.1 When to Lock with ROW SHARE MODE and ROW EXCLUSIVE MODE 15
8.4.2.2 When to Lock with SHARE MODE 16
8.4.2.3 When to Lock with SHARE ROW EXCLUSIVE MODE 17
8.4.2.4 When to Lock with EXCLUSIVE MODE 17
8.4.3 Letting Oracle Database Control Table Locking 17
8.4.4 Explicitly Acquiring Row Locks 18
8.4.5 Examples of Concurrency Under Explicit Locking 19
8.5 Using Oracle Lock Management Services (User Locks) 26
8.5.1 When to Use User Locks 26
8.5.2 Viewing and Monitoring Locks 27
8.6 Using Serializable Transactions for Concurrency Control 27
8.6.1 Transaction Interaction and Isolation Level 28
8.6.2 Setting Isolation Levels 30
8.6.3 Serializable Transactions and Referential Integrity 31
8.6.4 READ COMMITTED and SERIALIZABLE Isolation Levels 33
8.6.4.1 Transaction Set Consistency Differences 33
8.6.4.2 Choosing Transaction Isolation Levels 34
8.7 Nonblocking and Blocking DDL Statements 35
8.8 Autonomous Transactions 35
8.8.1 Examples of Autonomous Transactions 38
8.8.1.1 Ordering a Product 38
8.8.1.2 Withdrawing Money from a Bank Account 39
8.8.2 Declaring Autonomous Routines 41
8.9 Resuming Execution After Storage Allocation Errors 42
8.9.1 What Operations Have Resumable Storage Allocation? 42

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.9.2 Handling Suspended Storage Allocation 42
8.9.2.1 Using an AFTER SUSPEND Trigger in the Application 43
8.9.2.2 Checking for Suspended Statements 44
8.10 Using IF EXISTS and IF NOT EXISTS 45
8.10.1 Using IF NOT EXISTS with CREATE Command 45
8.10.2 Using IF EXISTS with ALTER Command 46
8.10.3 Using IF EXISTS with DROP Command 46
8.10.4 Supported Object Types 47
8.10.5 Limitations for CREATE OR REPLACE Statements 48
8.10.6 SQL*Plus Output Messages for DDL Statements 48

9 Using SQL Data Types in Database Applications

9.1 Using the Correct and Most Specific Data Type 1
9.1.1 How the Correct Data Type Increases Data Integrity 2
9.1.2 How the Most Specific Data Type Decreases Storage Requirements 2
9.1.3 How the Correct Data Type Improves Performance 3
9.2 Representing Character Data 6
9.3 Representing Numeric Data 7
9.3.1 Floating-Point Number Components 8
9.3.2 Floating-Point Number Formats 8
9.3.2.1 Binary Floating-Point Formats 9
9.3.3 Representing Special Values with Native Floating-Point Data Types 10
9.3.4 Comparing Native Floating-Point Values 11
9.3.5 Arithmetic Operations with Native Floating-Point Data Types 11
9.3.6 Conversion Functions for Native Floating-Point Data Types 12
9.3.7 Client Interfaces for Native Floating-Point Data Types 12
9.4 Representing Date and Time Data 13
9.4.1 Displaying Current Date and Time 14
9.4.2 Inserting and Displaying Dates 15
9.4.3 Inserting and Displaying Times 16
9.4.4 Arithmetic Operations with Datetime Data Types 17
9.4.5 Conversion Functions for Datetime Data Types 18
9.4.6 Importing, Exporting, and Comparing Datetime Types 19
9.5 Representing Specialized Data 19
9.5.1 Representing Spatial Data 19
9.5.2 Representing Large Amounts of Data 19
9.5.2.1 Large Objects (LOBs) 19
9.5.2.2 LONG and LONG RAW Data Types 20
9.5.3 Representing JSON Data 21
9.5.4 Representing Searchable Text 21
9.5.5 Representing XML Data 21

Copyright © 1999, 2025, Oracle and/or its affiliates.

9.5.6 Representing Dynamically Typed Data 22
9.5.7 Representing ANSI, DB2, and SQL/DS Data 24
9.6 Identifying Rows by Address 24
9.7 Displaying Metadata for SQL Operators and Functions 25
9.7.1 ARGn Data Type 26
9.7.2 DISP_TYPE Data Type 26
9.7.3 SQL Data Type Families 26

10 Registering Application Data Usage with the Database

10.1 Data Use Case Domains 1
10.1.1 Overview of Use Case Domains 2
10.1.2 Use Case Domain Types and When to Use Them 2
10.1.3 Privileges Required for Use Case Domains 3
10.1.4 Using a Single-column Use Case Domain 3
10.1.4.1 Creating a Use Case Domain 4
10.1.4.2 Associating Use Case Domains with Columns at Table Creation 6
10.1.4.3 Associating Use Case Domains with Existing or New Columns 10
10.1.4.4 Altering a Use Case Domain 12
10.1.4.5 Disassociating a Use Case Domain from a Column 14
10.1.4.6 Dropping a Use Case Domain 16
10.1.5 Using a Multi-column Use Case Domain 18
10.1.5.1 Creating a Multi-column Use Case Domain 18
10.1.5.2 Associating a Multi-column Use Case Domain at Table Creation 19
10.1.5.3 Associating a Multi-column Use Case Domain with Existing Columns 21
10.1.5.4 Altering a Multi-column Use Case Domain 22
10.1.5.5 Disassociating a Multi-column Use Case Domain from a Column 23
10.1.5.6 Dropping a Multi-column Use Case Domain 23
10.1.6 Using a Flexible Use Case Domain 24
10.1.6.1 Creating a Flexible Use Case Domain 24
10.1.6.2 Associating a Flexible Use Case Domain at Table Creation 26
10.1.6.3 Associating a Flexible Domain with Existing Columns 29
10.1.6.4 Disassociating a Flexible Use Case Domain from Columns 30
10.1.6.5 Dropping a Flexible Use Case Domain 30
10.1.7 Using an Enumeration Use Case Domain 31
10.1.7.1 About Enumeration Type 31
10.1.7.2 Enumeration Domains Overview 31
10.1.7.3 Creating an Enumeration Domain 32
10.1.7.4 Associating an Enumeration Domain at Table Creation 34
10.1.7.5 Associating an Enumeration Domain with Existing Columns 38
10.1.8 Specifying a Data Type for a Domain 38
10.1.9 Changing the Use Case Domain Properties 43

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.1.10 SQL Functions for Use Case Domains 47
10.1.11 Viewing Domain Information 48
10.1.11.1 Dictionary Views for Use Case Domains 48
10.1.12 Built-in Use Case Domains 49
10.2 Schema Annotations 88
10.2.1 Overview of Annotations 89
10.2.2 Annotations and Comments 90
10.2.3 Supported Database Objects 90
10.2.4 Privileges Required for Using Annotations 90
10.2.5 DDL Statements for Annotations 91
10.2.5.1 Annotation Syntax 91
10.2.5.2 DDL Statements to Annotate a Table 92
10.2.5.3 DDL Statements to Annotate a Table Column 93
10.2.5.4 DDL Statements to Annotate Views and Materialized Views 94
10.2.5.5 DDL Statements to Annotate Indexes 96
10.2.5.6 DDL Statements to Annotate Domains 96
10.2.5.7 Dictionary Table and Views 97

11 Using Regular Expressions in Database Applications

11.1 Overview of Regular Expressions 1
11.2 Oracle SQL Support for Regular Expressions 2
11.3 Oracle SQL and POSIX Regular Expression Standard 4
11.4 Operators in Oracle SQL Regular Expressions 5
11.4.1 POSIX Operators in Oracle SQL Regular Expressions 5
11.4.2 Oracle SQL Multilingual Extensions to POSIX Standard 8
11.4.3 Oracle SQL PERL-Influenced Extensions to POSIX Standard 9
11.5 Using Regular Expressions in SQL Statements: Scenarios 10
11.5.1 Using a Constraint to Enforce a Phone Number Format 10
11.5.2 Example: Enforcing a Phone Number Format with Regular Expressions 11
11.5.3 Example: Inserting Phone Numbers in Correct and Incorrect Formats 11
11.5.4 Using Back References to Reposition Characters 12

12 Using Indexes in Database Applications

12.1 Guidelines for Managing Indexes 1
12.2 Managing Indexes 2
12.3 When to Use Domain Indexes 2
12.4 When to Use Function-Based Indexes 2
12.4.1 Advantages of Function-Based Indexes 3
12.4.2 Disadvantages of Function-Based Indexes 4
12.4.3 Example: Function-Based Index for Precomputing Arithmetic Expression 6

Copyright © 1999, 2025, Oracle and/or its affiliates.

12.4.4 Example: Function-Based Indexes on Object Column 7
12.4.5 Example: Function-Based Index for Faster Case-Insensitive Searches 8
12.4.6 Example: Function-Based Index for Language-Dependent Sorting 8

13 Maintaining Data Integrity in Database Applications

13.1 Enforcing Business Rules with Constraints 2
13.2 Enforcing Business Rules with Both Constraints and Application Code 3
13.3 Creating Indexes for Use with Constraints 4
13.4 When to Use NOT NULL Constraints 4
13.5 When to Use Default Column Values 6
13.6 Choosing a Primary Key for a Table (PRIMARY KEY Constraint) 7
13.7 When to Use UNIQUE Constraints 8
13.8 Enforcing Referential Integrity with FOREIGN KEY Constraints 9
13.8.1 FOREIGN KEY Constraints and NULL Values 11
13.8.2 Defining Relationships Between Parent and Child Tables 11
13.8.3 Rules for Multiple FOREIGN KEY Constraints 12
13.8.4 Deferring Constraint Checks 12
13.9 Minimizing Space and Time Overhead for Indexes Associated with Constraints 14
13.10 Guidelines for Indexing Foreign Keys 15
13.11 Referential Integrity in a Distributed Database 15
13.12 When to Use CHECK Constraints 16
13.12.1 Restrictions on CHECK Constraints 16
13.12.2 Designing CHECK Constraints 17
13.12.3 Rules for Multiple CHECK Constraints 17
13.12.4 Choosing Between CHECK and NOT NULL Constraints 17
13.13 Using PRECHECK to Pre-validate a CHECK Constraint 18
13.13.1 PRECHECK Syntax and Definition 19
13.13.1.1 Supported Conditions for JSON Schema Validation 20
13.13.2 Enabling PRECHECK for a New Relational Table 26
13.13.3 Enabling PRECHECK for an Existing Table 29
13.13.4 Guidelines for Using PRECHECK 31
13.14 Examples of Defining Constraints 32
13.14.1 Privileges Needed to Define Constraints 33
13.14.2 Naming Constraints 34
13.15 Enabling and Disabling Constraints 34
13.15.1 Why Disable Constraints? 34
13.15.2 Creating Enabled Constraints (Default) 35
13.15.3 Creating Disabled Constraints 35
13.15.4 Enabling Existing Constraints 36
13.15.5 Disabling Existing Constraints 36
13.15.6 Guidelines for Enabling and Disabling Key Constraints 37

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.15.7 Fixing Constraint Exceptions 37
13.16 Modifying Constraints 38
13.17 Renaming Constraints 39
13.18 Dropping Constraints 40
13.19 Managing FOREIGN KEY Constraints 40
13.19.1 Data Types and Names for Foreign Key Columns 41
13.19.2 Limit on Columns in Composite Foreign Keys 41
13.19.3 Foreign Key References Primary Key by Default 41
13.19.4 Privileges Required to Create FOREIGN KEY Constraints 41
13.19.5 Choosing How Foreign Keys Enforce Referential Integrity 41
13.20 Viewing Information About Constraints 42

Part III PL/SQL for Application Developers

14 Coding PL/SQL Subprograms and Packages

14.1 Overview of PL/SQL Subprograms 1
14.2 Overview of PL/SQL Packages 2
14.3 Overview of PL/SQL Units 3
14.3.1 PLSQL_OPTIMIZE_LEVEL Compilation Parameter 4
14.4 Creating PL/SQL Subprograms and Packages 6
14.4.1 Privileges Needed to Create Subprograms and Packages 7
14.4.2 Creating Subprograms and Packages 7
14.4.3 PL/SQL Object Size Limits 8
14.4.4 PL/SQL Data Types 9
14.4.4.1 PL/SQL Scalar Data Types 9
14.4.4.2 PL/SQL Composite Data Types 11
14.4.4.3 Abstract Data Types 12
14.4.5 Returning Result Sets to Clients 12
14.4.5.1 Advantages of Cursor Variables 13
14.4.5.2 Disadvantages of Cursor Variables 14
14.4.5.3 Returning Query Results Implicitly 16
14.4.6 Returning Large Amounts of Data from a Function 17
14.4.7 PL/SQL Function Result Cache 17
14.4.8 Overview of Bulk Binding 18
14.4.8.1 DML Statements that Reference Collections 19
14.4.8.2 SELECT Statements that Reference Collections 19
14.4.8.3 FOR Loops that Reference Collections and Return DML 20
14.4.9 PL/SQL Dynamic SQL 21
14.5 Altering PL/SQL Subprograms and Packages 21
14.6 Deprecating Packages, Subprograms, and Types 22

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.7 Dropping PL/SQL Subprograms and Packages 22
14.8 Compiling PL/SQL Units for Native Execution 23
14.9 Invoking Stored PL/SQL Subprograms 23
14.9.1 Privileges Required to Invoke a Stored Subprogram 24
14.9.2 Invoking a Subprogram Interactively from Oracle Tools 25
14.9.3 Invoking a Subprogram from Another Subprogram 26
14.9.4 Invoking a Remote Subprogram 27
14.9.4.1 Synonyms for Remote Subprograms 28
14.9.4.2 Transactions That Invoke Remote Subprograms 29
14.10 Invoking Stored PL/SQL Functions from SQL Statements 30
14.10.1 Why Invoke PL/SQL Functions from SQL Statements? 31
14.10.2 Where PL/SQL Functions Can Appear in SQL Statements 31
14.10.3 When PL/SQL Functions Can Appear in SQL Expressions 32
14.10.4 Controlling Side Effects of PL/SQL Functions Invoked from SQL Statements 32
14.10.4.1 Restrictions on Functions Invoked from SQL Statements 33
14.10.4.2 PL/SQL Functions Invoked from Parallelized SQL Statements 34
14.10.4.3 PRAGMA RESTRICT_REFERENCES 35
14.11 Analyzing and Debugging Stored Subprograms 38
14.11.1 PL/Scope 38
14.11.2 PL/SQL Hierarchical Profiler 39
14.11.3 Debugging PL/SQL and Java 39
14.11.3.1 Compiling Code for Debugging 39
14.11.3.2 Privileges for Debugging PL/SQL and Java Stored Subprograms 40
14.12 Package Invalidations and Session State 41
14.13 Example: Raising an ORA-04068 Error 41
14.14 Example: Trapping ORA-04068 42

15 Using PL/Scope

15.1 Overview of PL/Scope 1
15.2 Privileges Required for Using PL/Scope 2
15.3 Specifying Identifier and Statement Collection 2
15.4 How Much Space is PL/Scope Data Using? 3
15.5 Viewing PL/Scope Data 4
15.5.1 Static Data Dictionary Views for PL/SQL and SQL Identifiers 4
15.5.1.1 PL/SQL and SQL Identifier Types that PL/Scope Collects 4
15.5.1.2 About Identifiers Usages 6
15.5.1.3 Identifiers Usage Unique Keys 7
15.5.1.4 About Identifiers Usage Context 8
15.5.1.5 About Identifiers Signature 10
15.5.2 Static Data Dictionary Views for SQL Statements 12
15.5.2.1 SQL Statement Types that PL/Scope Collects 12

Copyright © 1999, 2025, Oracle and/or its affiliates.

15.5.2.2 Statements Location Unique Keys 13
15.5.2.3 About SQL Statement Usage Context 14
15.5.2.4 About SQL Statements Signature 15
15.5.3 SQL Developer 16
15.6 Overview of Data Dictionary Views Useful to Manage PL/SQL Code 16
15.7 Sample PL/Scope Session 17

16 Using the PL/SQL Hierarchical Profiler

16.1 Overview of PL/SQL Hierarchical Profiler 1
16.2 Collecting Profile Data 2
16.3 Understanding Raw Profiler Output 4
16.3.1 Namespaces of Tracked Subprograms 7
16.3.2 Special Function Names 7
16.4 Analyzing Profile Data 8
16.4.1 Creating Hierarchical Profiler Tables 8
16.4.2 Understanding Hierarchical Profiler Tables 9
16.4.2.1 Hierarchical Profiler Database Table Columns 10
16.4.2.2 Distinguishing Between Overloaded Subprograms 12
16.4.2.3 Hierarchical Profiler Tables for Sample PL/SQL Procedure 12
16.4.2.4 Examples of Calls to DBMS_HPROF.analyze with Options 13
16.5 plshprof Utility 15
16.5.1 plshprof Options 15
16.5.2 HTML Report from a Single Raw Profiler Output File 16
16.5.2.1 First Page of Report 16
16.5.2.2 Function-Level Reports 17
16.5.2.3 Module-Level Reports 18
16.5.2.4 Namespace-Level Reports 19
16.5.2.5 Parents and Children Report for a Function 19
16.5.2.6 Understanding PL/SQL Hierarchical Profiler SQL-Level Reports 21
16.5.3 HTML Difference Report from Two Raw Profiler Output Files 21
16.5.3.1 Difference Report Conventions 22
16.5.3.2 First Page of Difference Report 22
16.5.3.3 Function-Level Difference Reports 23
16.5.3.4 Module-Level Difference Reports 25
16.5.3.5 Namespace-Level Difference Reports 25
16.5.3.6 Parents and Children Difference Report for a Function 25

17 Using PL/SQL Basic Block Coverage to Maintain Quality

17.1 Overview of PL/SQL Basic Block Coverage 1
17.2 Collecting PL/SQL Code Coverage Data 2

Copyright © 1999, 2025, Oracle and/or its affiliates.

17.3 PL/SQL Code Coverage Tables Description 2

18 Developing PL/SQL Web Applications

18.1 Overview of PL/SQL Web Applications 1
18.2 Implementing PL/SQL Web Applications 2
18.2.1 PL/SQL Gateway 2
18.2.1.1 mod_plsql 2
18.2.1.2 Embedded PL/SQL Gateway 3
18.2.2 PL/SQL Web Toolkit 3
18.3 Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application 4
18.4 Using Embedded PL/SQL Gateway 4
18.4.1 How Embedded PL/SQL Gateway Processes Client Requests 5
18.4.2 Installing Embedded PL/SQL Gateway 6
18.4.3 Configuring Embedded PL/SQL Gateway 7
18.4.3.1 Configuring Embedded PL/SQL Gateway: Overview 7
18.4.3.2 Configuring User Authentication for Embedded PL/SQL Gateway 9
18.4.4 Invoking PL/SQL Stored Subprograms Through Embedded PL/SQL Gateway 18
18.4.5 Securing Application Access with Embedded PL/SQL Gateway 18
18.4.6 Restrictions in Embedded PL/SQL Gateway 19
18.4.7 Using Embedded PL/SQL Gateway: Scenario 19
18.5 Generating HTML Output with PL/SQL 21
18.6 Passing Parameters to PL/SQL Web Applications 22
18.6.1 Passing List and Dropdown-List Parameters from an HTML Form 22
18.6.2 Passing Option and Check Box Parameters from an HTML Form 23
18.6.3 Passing Entry-Field Parameters from an HTML Form 23
18.6.4 Passing Hidden Parameters from an HTML Form 25
18.6.5 Uploading a File from an HTML Form 25
18.6.6 Submitting a Completed HTML Form 25
18.6.7 Handling Missing Input from an HTML Form 26
18.6.8 Maintaining State Information Between Web Pages 26
18.7 Performing Network Operations in PL/SQL Subprograms 27
18.7.1 Internet Protocol Version 6 (IPv6) Support 27
18.7.2 Sending E-Mail from PL/SQL 27
18.7.3 Getting a Host Name or Address from PL/SQL 28
18.7.4 Using TCP/IP Connections from PL/SQL 28
18.7.5 Retrieving HTTP URL Contents from PL/SQL 29
18.7.6 Using Tables, Image Maps, Cookies, and CGI Variables from PL/SQL 31

Copyright © 1999, 2025, Oracle and/or its affiliates.

19 Using Continuous Query Notification (CQN)

19.1 About Object Change Notification (OCN) 2
19.2 About Query Result Change Notification (QRCN) 2
19.2.1 Guaranteed Mode 3
19.2.2 Best-Effort Mode 3
19.2.2.1 Example: Query Too Complex for QRCN in Guaranteed Mode 4
19.2.2.2 Example: Query Whose Simplified Version Invalidates Objects 4
19.3 Events that Generate Notifications 5
19.3.1 Committed DML Transactions 5
19.3.2 Committed DDL Statements 6
19.3.3 Deregistration 7
19.3.4 Global Events 7
19.4 Notification Contents 7
19.5 Good Candidates for CQN 8
19.6 Creating CQN Registrations 11
19.7 Using PL/SQL to Create CQN Registrations 11
19.7.1 PL/SQL CQN Registration Interface 12
19.7.2 CQN Registration Options 12
19.7.2.1 Notification Type Option 13
19.7.2.2 QRCN Mode (QRCN Notification Type Only) 13
19.7.2.3 ROWID Option 13
19.7.2.4 Operations Filter Option (OCN Notification Type Only) 14
19.7.2.5 Transaction Lag Option (OCN Notification Type Only) 15
19.7.2.6 Notification Grouping Options 15
19.7.2.7 Reliable Option 16
19.7.2.8 Purge-on-Notify and Timeout Options 16
19.7.3 Prerequisites for Creating CQN Registrations 16
19.7.4 Queries that Can Be Registered for Object Change Notification (OCN) 17
19.7.5 Queries that Can Be Registered for Query Result Change Notification (QRCN) 17
19.7.5.1 Queries that Can Be Registered for QRCN in Guaranteed Mode 18
19.7.5.2 Queries that Can Be Registered for QRCN Only in Best-Effort Mode 19
19.7.5.3 Queries that Cannot Be Registered for QRCN in Either Mode 20
19.7.6 Using PL/SQL to Register Queries for CQN 20
19.7.6.1 Creating a PL/SQL Notification Handler 21
19.7.6.2 Creating a CQ_NOTIFICATION$_REG_INFO Object 21
19.7.6.3 Identifying Individual Queries in a Notification 24
19.7.6.4 Adding Queries to an Existing Registration 25
19.7.7 Best Practices for CQN Registrations 25
19.7.8 Troubleshooting CQN Registrations 26
19.7.9 Deleting Registrations 27
19.7.10 Configuring CQN: Scenario 27

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.7.10.1 Creating a PL/SQL Notification Handler 27
19.7.10.2 Registering the Queries 29
19.8 Using OCI to Create CQN Registrations 31
19.8.1 Using OCI for Query Result Set Notifications 31
19.8.2 Using OCI to Register a Continuous Query Notification 32
19.8.3 Using OCI for Client Initiated CQN Registrations 33
19.8.4 Using OCI Subscription Handle Attributes for Continuous Query Notification 34
19.8.5 OCI_ATTR_CQ_QUERYID Attribute 36
19.8.6 Using OCI Continuous Query Notification Descriptors 36
19.8.6.1 OCI_DTYPE_CHDES 36
19.8.7 Demonstrating Continuous Query Notification in an OCI Sample Program 38
19.9 Querying CQN Registrations 47
19.10 Interpreting Notifications 48
19.10.1 Interpreting a CQ_NOTIFICATION$_DESCRIPTOR Object 48
19.10.2 Interpreting a CQ_NOTIFICATION$_TABLE Object 49
19.10.3 Interpreting a CQ_NOTIFICATION$_QUERY Object 50
19.10.4 Interpreting a CQ_NOTIFICATION$_ROW Object 50

Part IV Advanced Topics for Application Developers

20 Choosing a Programming Environment

20.1 Overview of Application Architecture 2
20.1.1 Client/Server Architecture 2
20.1.2 Server-Side Programming 2
20.1.3 Two-Tier and Three-Tier Architecture 3
20.2 Overview of the Program Interface 3
20.2.1 User Interface 3
20.2.2 Stateful and Stateless User Interfaces 4
20.3 Overview of PL/SQL 4
20.4 Overview of Oracle Database Java Support 4
20.4.1 Overview of Oracle JVM 5
20.4.2 Overview of Oracle JDBC 6
20.4.2.1 Oracle JDBC Drivers 6
20.4.2.2 Sample JDBC 2.0 Program 8
20.4.2.3 Sample Pre-2.0 JDBC Program 8
20.4.3 Overview of Oracle SQLJ 9
20.4.3.1 Benefits of SQLJ 10
20.4.4 Comparison of Oracle JDBC and Oracle SQLJ 11
20.4.5 Overview of Java Stored Subprograms 11
20.4.6 Overview of Oracle Database Web Services 12

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.5 Overview of JavaScript 13
20.5.1 Multilingual Engine Overview 13
20.5.2 MLE Concepts 14
20.5.3 Understanding MLE Execution Context and Runtime Isolation 14
20.5.4 MLE Environment Overview 15
20.5.5 JavaScript MLE Modules Overview 16
20.5.6 JavaScript MLE Call Specification Overview 17
20.5.7 Invoking JavaScript in the Database 17
20.5.8 Invoking JavaScript Using MLE Modules 18
20.5.8.1 Using MLE Module Contexts 18
20.5.8.2 Specifying an Environment for Call Specifications 19
20.5.8.3 Managing JavaScript MLE Modules 19
20.5.8.4 Running JavaScript Code Using MLE Modules 20
20.5.9 Invoking JavaScript Using Dynamic MLE Execution 22
20.5.9.1 Dynamic MLE Execution Overview 23
20.5.9.2 Using Dynamic MLE Execution contexts 23
20.5.9.3 Specifying an Environment for Dynamic MLE Contexts 24
20.5.9.4 Running JavaScript Code Using Dynamic MLE Execution 24
20.5.10 Privileges for Working with JavaScript in MLE 26
20.5.10.1 MLE User Privileges 26
20.5.11 Other Supported MLE Features 27
20.6 Choosing PL/SQL, Java, or JavaScript 28
20.6.1 Similarities of PL/SQL, Java, and JavaScript 31
20.6.2 Advantages of PL/SQL 31
20.6.3 Advantages of Java 32
20.6.4 Advantages of JavaScript 33
20.7 Overview of Precompilers 33
20.7.1 Overview of the Pro*C/C++ Precompiler 34
20.7.2 Overview of the Pro*COBOL Precompiler 36
20.8 Overview of OCI and OCCI 37
20.8.1 Advantages of OCI and OCCI 38
20.8.2 OCI and OCCI Functions 38
20.8.3 Procedural and Nonprocedural Elements of OCI and OCCI Applications 39
20.8.4 Building an OCI or OCCI Application 39
20.9 Comparison of Precompilers and OCI 40
20.10 Overview of Oracle Data Provider for .NET (ODP.NET) 41
20.11 Overview of OraOLEDB 42

21 Developing Applications with Multiple Programming Languages

21.1 Overview of Multilanguage Programs 1
21.2 What Is an External Procedure? 3

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.3 Overview of Call Specification for External Procedures 3
21.4 Loading External Procedures 4
21.4.1 Define the C Procedures 5
21.4.2 Set Up the Environment 6
21.4.3 Identify the DLL 8
21.4.4 Publish the External Procedures 9
21.5 Publishing External Procedures 9
21.5.1 AS LANGUAGE Clause for Java Class Methods 11
21.5.2 AS LANGUAGE Clause for External C Procedures 11
21.5.2.1 LIBRARY 11
21.5.2.2 NAME 11
21.5.2.3 LANGUAGE 11
21.5.2.4 CALLING STANDARD 11
21.5.2.5 WITH CONTEXT 12
21.5.2.6 PARAMETERS 12
21.5.2.7 AGENT IN 12
21.6 Publishing Java Class Methods 12
21.7 Publishing External C Procedures 13
21.8 Locations of Call Specifications 13
21.8.1 Example: Locating a Call Specification in a PL/SQL Package 14
21.8.2 Example: Locating a Call Specification in a PL/SQL Package Body 14
21.8.3 Example: Locating a Call Specification in an ADT Specification 14
21.8.4 Example: Locating a Call Specification in an ADT Body 15
21.8.5 Example: Java with AUTHID 15
21.8.6 Example: C with Optional AUTHID 15
21.8.7 Example: Mixing Call Specifications in a Package 15
21.9 Passing Parameters to External C Procedures with Call Specifications 16
21.9.1 Specifying Data Types 17
21.9.2 External Data Type Mappings 19
21.9.3 Passing Parameters BY VALUE or BY REFERENCE 21
21.9.4 Declaring Formal Parameters 22
21.9.5 Overriding Default Data Type Mapping 22
21.9.6 Specifying Properties 23
21.9.6.1 INDICATOR 24
21.9.6.2 LENGTH and MAXLEN 25
21.9.6.3 CHARSETID and CHARSETFORM 25
21.9.6.4 Repositioning Parameters 26
21.9.6.5 SELF 26
21.9.6.6 BY REFERENCE 28
21.9.6.7 WITH CONTEXT 28
21.9.6.8 Interlanguage Parameter Mode Mappings 29
21.10 Running External Procedures with CALL Statements 29

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.10.1 Preconditions for External Procedures 30
21.10.1.1 Privileges of External Procedures 30
21.10.1.2 Managing Permissions 31
21.10.1.3 Creating Synonyms for External Procedures 31
21.10.2 CALL Statement Syntax 31
21.10.3 Calling Java Class Methods 32
21.10.4 Calling External C Procedures 32
21.11 Handling Errors and Exceptions in Multilanguage Programs 33
21.12 Using Service Routines with External C Procedures 33
21.12.1 OCIExtProcAllocCallMemory 33
21.12.2 OCIExtProcRaiseExcp 37
21.12.3 OCIExtProcRaiseExcpWithMsg 38
21.13 Doing Callbacks with External C Procedures 39
21.13.1 OCIExtProcGetEnv 39
21.13.2 Object Support for OCI Callbacks 41
21.13.3 Restrictions on Callbacks 41
21.13.4 Debugging External C Procedures 42
21.13.5 Example: Calling an External C Procedure 43
21.13.6 Global Variables in External C Procedures 43
21.13.7 Static Variables in External C Procedures 43
21.13.8 Restrictions on External C Procedures 44

22 Using Oracle Flashback Technology

22.1 Overview of Oracle Flashback Technology 1
22.1.1 Application Development Features 2
22.1.2 Database Administration Features 4
22.2 Configuring Your Database for Oracle Flashback Technology 4
22.2.1 Configuring Your Database for Automatic Undo Management 5
22.2.2 Configuring Your Database for Oracle Flashback Transaction Query 6
22.2.3 Configuring Your Database for Flashback Transaction 6
22.2.4 Enabling Oracle Flashback Operations on Specific LOB Columns 7
22.2.5 Granting Necessary Privileges 7
22.3 Using Oracle Flashback Query (SELECT AS OF) 8
22.3.1 Example: Examining and Restoring Past Data 9
22.3.2 Guidelines for Oracle Flashback Query 9
22.4 Using Oracle Flashback Version Query 10
22.5 Using Oracle Flashback Transaction Query 12
22.6 Using Oracle Flashback Transaction Query with Oracle Flashback Version Query 13
22.7 Using DBMS_FLASHBACK Package 15
22.7.1 Using Flashback Version Query with DBMS_FLASHBACK 16
22.8 Using Flashback Transaction 17

Copyright © 1999, 2025, Oracle and/or its affiliates.

22.8.1 Dependent Transactions 18
22.8.2 TRANSACTION_BACKOUT Parameters 18
22.8.3 TRANSACTION_BACKOUT Reports 19
22.8.3.1 *_FLASHBACK_TXN_STATE 19
22.8.3.2 *_FLASHBACK_TXN_REPORT 20
22.9 Using Flashback Time Travel 20
22.9.1 DDL Statements on Tables Enabled for Flashback Archive 22
22.9.2 Creating a Flashback Archive 23
22.9.3 Altering a Flashback Archive 24
22.9.4 Dropping a Flashback Archive 25
22.9.5 Specifying the Default Flashback Archive 26
22.9.6 Enabling and Disabling Flashback Archive 26
22.9.7 Viewing Flashback Archive Data 27
22.9.8 Transporting Flashback Archive Data between Databases 28
22.9.9 Flashback Time Travel Scenarios 28
22.9.9.1 Scenario: Using Flashback Time Travel to Enforce Digital Shredding 28
22.9.9.2 Scenario: Using Flashback Time Travel to Access Historical Data 28
22.9.9.3 Scenario: Using Flashback Time Travel to Generate Reports 29
22.9.9.4 Scenario: Using Flashback Time Travel for Auditing 29
22.9.9.5 Scenario: Using Flashback Time Travel to Recover Data 29
22.9.10 Protecting Flashback Archive Data 30
22.10 General Guidelines for Oracle Flashback Technology 33
22.11 Oracle Virtual Private Database Policies and Oracle Flashback Time Travel 35
22.12 Performance Guidelines for Oracle Flashback Technology 38
22.13 Multitenant Container Database Restrictions for Oracle Flashback Technology 38

23 Developing Applications with the Publish-Subscribe Model

23.1 Introduction to the Publish-Subscribe Model 1
23.2 Publish-Subscribe Architecture 2
23.2.1 Database Events 2
23.2.2 Oracle Advanced Queuing 2
23.2.3 Client Notification 2
23.3 Publish-Subscribe Concepts 3
23.4 Examples of a Publish-Subscribe Mechanism 5

24 Using the Oracle Database ODBC Driver

Copyright © 1999, 2025, Oracle and/or its affiliates.

25 Using the Identity Code Package

25.1 Identity Concepts 1
25.2 What Is the Identity Code Package? 5
25.3 Using the Identity Code Package 6
25.3.1 Storing RFID Tags in Oracle Database Using MGD_ID ADT 6
25.3.1.1 Creating a Table with MGD_ID Column Type and Storing EPC Tag Encodings in the Column 6
25.3.1.2 Constructing MGD_ID Objects to Represent RFID Tags 7
25.3.1.3 Inserting an MGD_ID Object into a Database Table 9
25.3.1.4 Querying MGD_ID Column Type 10
25.3.2 Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type 10
25.3.3 Using MGD_ID ADT Functions 11
25.3.3.1 Using the get_component Function with the MGD_ID Object 11
25.3.3.2 Parsing Tag Data from Standard Representations 11
25.3.3.3 Reconstructing Tag Representations from Fields 13
25.3.3.4 Translating Between Tag Representations 13
25.3.4 Defining a Category of Identity Codes and Adding Encoding Schemes to an Existing Category 13
25.3.4.1 Creating a Category of Identity Codes 14
25.3.4.2 Adding Two Metadata Schemes to a Newly Created Category 14
25.4 Identity Code Package Types 18
25.5 DBMS_MGD_ID_UTL Package 19
25.6 Identity Code Metadata Tables and Views 20
25.7 Electronic Product Code (EPC) Concepts 22
25.7.1 RFID Technology and EPC v1.1 Coding Schemes 22
25.7.2 Product Code Concepts and Their Current Use 23
25.7.2.1 Electronic Product Code (EPC) 23
25.7.2.2 Global Trade Identification Number (GTIN) and Serializable Global Trade Identification Number (SGTIN) 24
25.7.2.3 Serial Shipping Container Code (SSCC) 25
25.7.2.4 Global Location Number (GLN) and Serializable Global Location Number (SGLN) 25
25.7.2.5 Global Returnable Asset Identifier (GRAI) 25
25.7.2.6 Global Individual Asset Identifier (GIAI) 25
25.7.2.7 RFID EPC Network 25
25.8 Oracle Database Tag Data Translation Schema 25

26 Microservices Architecture

26.1 About Microservices Architecture 1
26.2 Features of Microservices Architecture 3
26.3 Challenges in a Distributed System 3

Copyright © 1999, 2025, Oracle and/or its affiliates.

26.4 Solutions for Microservices 4
26.4.1 Two-Phase Commit Pattern 5
26.4.2 Saga Design Pattern 6
26.4.2.1 Why Use Sagas? 6
26.4.2.2 Saga Implementation Approaches 7
26.4.2.3 Successful and Unsuccessful Sagas 7
26.4.2.4 Saga Flow 7
26.4.3 Backend as a Service For The 12 Patterns For Microservices Success 8

27 Oracle Backend for Microservices and AI

27.1 About Oracle Backend for Microservices and AI 1
27.2 Getting Started with Oracle Backend for Microservices and AI 1
27.2.1 CloudBank Sample Application 3

28 Developing Applications with Sagas

28.1 Implementing Sagas with Oracle Database 1
28.2 Oracle Saga Framework Overview 2
28.3 Saga Framework Features 2
28.4 Saga Framework Concepts 3
28.5 Initializing the Saga Framework 6
28.6 Setting Up a Saga Topology 6
28.6.1 Adding a Message Broker 7
28.6.2 Adding a Coordinator 7
28.6.3 Adding a Participant 8
28.6.4 Managing Participants and Message Brokers 8
28.6.5 Message Propagation 9
28.6.6 About Dictionary Tables 9
28.6.7 Example: Saga Framework Setup 10
28.7 Managing a Saga Using the PL/SQL Interface 11
28.7.1 Example: Saga PL/SQL Program 12
28.8 Developing Java Applications Using Saga Annotations 13
28.8.1 LRA and Saga Annotations 14
28.8.2 Packaging 19
28.8.3 Configuration 19
28.8.4 Saga Interface and Classes 21
28.8.4.1 Saga Interface 21
28.8.4.2 SagaMessageContext Class 30
28.8.4.3 SagaParticipant Class 30
28.8.4.4 SagaInitiator Class 33
28.8.5 Example Program 35

Copyright © 1999, 2025, Oracle and/or its affiliates.

28.9 Finalizing a Saga Explicitly 39
28.9.1 PL/SQL Callbacks for a PL/SQL Client 40
28.9.2 Integration with Lock-Free Reservation 42
28.10 AfterSaga Callbacks 43

29 Using Lock-Free Reservation

29.1 About Concurrency in Transaction Processing 1
29.2 Lock-Free Reservation Terminology 2
29.3 Lock-Free Reservation 3
29.3.1 Comparing Optimistic Locking and Lock-Free Reservation 5
29.3.2 Creating a Reservable Column at Table Creation 5
29.3.2.1 Reservation Journal Table Columns 6
29.3.3 Adding or Modifying Reservable Columns 7
29.3.4 About CHECK Constraints in Reservable Columns 8
29.3.5 Example: Conventional Locking and Lock-Free Reservation 9
29.3.6 Querying Reservable Column Views 10
29.4 Benefits of Using Lock-Free Reservation 11
29.5 Guidelines and Restrictions for Lock-Free Reservation 11
29.5.1 Guidelines and Restrictions for Reservable Columns 11
29.5.2 Guidelines and Restrictions for Update Statements 12
29.5.3 Guidelines for Inserts and Deletes 13
29.5.4 Guidelines for Concurrent DDL Statements 13
29.5.5 Restrictions for Reservation Journal Table 14

30 Developing Applications with Oracle XA

30.1 X/Open Distributed Transaction Processing (DTP) 1
30.1.1 DTP Terminology 2
30.1.2 Required Public Information 5
30.2 Oracle XA Library Subprograms 5
30.2.1 Oracle XA Library Subprograms 6
30.2.2 Oracle XA Interface Extensions 6
30.3 Developing and Installing XA Applications 7
30.3.1 DBA or System Administrator Responsibilities 7
30.3.2 Application Developer Responsibilities 8
30.3.3 Defining the xa_open String 8
30.3.3.1 Syntax of the xa_open String 8
30.3.3.2 Required Fields for the xa_open String 9
30.3.3.3 Optional Fields for the xa_open String 10
30.3.4 Using Oracle XA with Precompilers 12
30.3.4.1 Using Precompilers with the Default Database 12

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.3.4.2 Using Precompilers with a Named Database 13
30.3.5 Using Oracle XA with OCI 14
30.3.6 Managing Transaction Control with Oracle XA 14
30.3.7 Examples of Precompiler Applications 15
30.3.8 Migrating Precompiler or OCI Applications to TPM Applications 16
30.3.9 Managing Oracle XA Library Thread Safety 17
30.3.9.1 Specifying Threading in the Open String 17
30.3.9.2 Restrictions on Threading in Oracle XA 17
30.3.10 Using the DBMS_XA Package 17
30.4 Troubleshooting XA Applications 20
30.4.1 Accessing Oracle XA Trace Files 20
30.4.1.1 xa_open String DbgFl 21
30.4.1.2 Trace File Locations 21
30.4.2 Managing In-Doubt or Pending Oracle XA Transactions 22
30.4.3 Using SYS Account Tables to Monitor Oracle XA Transactions 22
30.5 Oracle XA Issues and Restrictions 23
30.5.1 Using Database Links in Oracle XA Applications 23
30.5.2 Managing Transaction Branches in Oracle XA Applications 23
30.5.3 Using Oracle XA with Oracle Real Application Clusters (Oracle RAC) 24
30.5.3.1 Oracle RAC XA Limitations 24
30.5.3.2 GLOBAL_TXN_PROCESSES Initialization Parameter 24
30.5.3.3 Managing Transaction Branches on Oracle RAC 25
30.5.3.4 Managing Instance Recovery in Oracle RAC with DTP Services (10.2) 26
30.5.3.5 Global Uniqueness of XIDs in Oracle RAC 27
30.5.3.6 Tight and Loose Coupling 27
30.5.4 SQL-Based Oracle XA Restrictions 28
30.5.4.1 Rollbacks and Commits 28
30.5.4.2 DDL Statements 28
30.5.4.3 Session State 28
30.5.4.4 EXEC SQL 28
30.5.5 Miscellaneous Restrictions 29

31 Developing Applications with Sessionless Transactions

31.1 Introduction to Managing Transactions and Sessionless Transactions 1
31.2 Sessionless Transactions Overview 3
31.3 Sessionless Transactions Capabilities 4
31.3.1 Example Use Cases 8
31.4 Benefits of Sessionless Transactions 12
31.5 Using Sessionless Transactions 13
31.5.1 Understanding Active Sessionless Transactions 14
31.5.2 Understanding the Lifecycle of Sessionless Transactions 14

Copyright © 1999, 2025, Oracle and/or its affiliates.

31.5.3 Understanding Server Round Trips and Pre-call and Post-call Functions 14
31.5.4 Prerequisites for Using Sessionless Transactions 16
31.5.5 Setting a Global Transaction ID 16
31.5.6 Starting a New Sessionless Transaction 18
31.5.6.1 Running SQL Statements within Sessionless Transactions 22
31.5.7 Retrieving a Global Transaction ID 22
31.5.8 Suspending a Sessionless Transaction 24
31.5.8.1 Time-out of a Suspended Sessionless Transaction 27
31.5.9 Resuming a Suspended Sessionless Transaction 29
31.5.10 Finalizing a Sessionless Transaction 33
31.5.10.1 Committing a Sessionless Transaction 33
31.5.10.2 Rolling Back a Sessionless Transaction 36
31.5.11 Example: Sessionless Transactions with OCI API 40
31.5.12 Rules and Guidelines for Using Sessionless Transactions 46
31.5.13 Error Messages and Notifications 48
31.6 Sessionless Transactions and Oracle Coordinated Distributed Transaction Interoperability 49
31.7 Restrictions for Sessionless Transactions 50
31.8 Draining Sessionless Transactions 51

32 Understanding Schema Object Dependency

32.1 Overview of Schema Object Dependency 1
32.1.1 Example: Displaying Dependent and Referenced Object Types 1
32.1.2 Example: Schema Object Change that Invalidates Some Dependents 2
32.1.3 Example: View That Depends on Multiple Objects 3
32.2 Querying Object Dependencies 4
32.3 Object Status 4
32.4 Invalidation of Dependent Objects 5
32.4.1 Session State and Referenced Packages 8
32.4.2 Security Authorization 8
32.5 Guidelines for Reducing Invalidation 8
32.5.1 Add Items to End of Package 8
32.5.2 Reference Each Table Through a View 9
32.6 Object Revalidation 9
32.6.1 Revalidation of Objects that Compiled with Errors 9
32.6.2 Revalidation of Unauthorized Objects 10
32.6.3 Revalidation of Invalid SQL Objects 10
32.6.4 Revalidation of Invalid PL/SQL Objects 10
32.7 Name Resolution in Schema Scope 10
32.8 Local Dependency Management 11
32.9 Remote Dependency Management 11

Copyright © 1999, 2025, Oracle and/or its affiliates.

32.9.1 Dependencies Among Local and Remote Database Procedures 12
32.9.2 Dependencies Among Other Remote Objects 12
32.9.3 Dependencies of Applications 12
32.10 Remote Procedure Call (RPC) Dependency Management 13
32.10.1 Time-Stamp Dependency Mode 13
32.10.2 RPC-Signature Dependency Mode 14
32.10.2.1 Changing Names and Default Values of Parameters 15
32.10.2.2 Changing Specification of Parameter Mode IN 16
32.10.2.3 Changing Subprogram Body 16
32.10.2.4 Changing Data Type Classes of Parameters 16
32.10.2.5 Changing Package Types 18
32.10.3 Controlling Dependency Mode 18
32.10.3.1 Dependency Resolution 19
32.10.3.2 Suggestions for Managing Dependencies 20
32.11 Shared SQL Dependency Management 20

33 Using Edition-Based Redefinition

33.1 Overview of Edition-Based Redefinition 1
33.2 Editions 2
33.2.1 Editioned and Noneditioned Objects 2
33.2.1.1 Name Resolution for Editioned and Noneditioned Objects 3
33.2.1.2 Noneditioned Objects That Can Depend on Editioned Objects 4
33.2.1.3 Editionable and Noneditionable Schema Object Types 5
33.2.1.4 Enabling Editions for a User 6
33.2.1.5 EDITIONABLE and NONEDITIONABLE Properties 9
33.2.1.6 Rules for Editioned Objects 10
33.2.2 Creating an Edition 11
33.2.3 Editioned Objects and Copy-on-Change 11
33.2.3.1 Example: Editioned Objects and Copy-on-Change 12
33.2.3.2 Example: Dropping an Editioned Object 13
33.2.3.3 Example: Creating an Object with the Name of a Dropped Inherited Object 15
33.2.4 Making an Edition Available to Some Users 16
33.2.5 Making an Edition Available to All Users 16
33.2.6 Current Edition and Session Edition 17
33.2.6.1 Your Initial Session Edition 17
33.2.6.2 Changing Your Session Edition 18
33.2.6.3 Displaying the Names of the Current and Session Editions 19
33.2.6.4 When the Current Edition Might Differ from the Session Edition 19
33.2.6.5 Disassociating a Session from an Edition 21
33.2.7 Retiring an Edition 21

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.2.8 Dropping an Edition 22
33.3 Editions and Audit Policies 24
33.4 Editioning Views 24
33.4.1 Creating an Editioning View 25
33.4.2 Partition-Extended Editioning View Names 25
33.4.3 Changing the Writability of an Editioning View 26
33.4.4 Replacing an Editioning View 26
33.4.5 Dropped or Renamed Base Tables 26
33.4.6 Adding Indexes and Constraints to the Base Table 26
33.4.7 SQL Optimizer Index Hints 27
33.5 Crossedition Triggers 27
33.5.1 Forward Crossedition Triggers 28
33.5.2 Reverse Crossedition Triggers 28
33.5.3 Crossedition Trigger Interaction with Editions 28
33.5.3.1 Which Triggers Are Visible 28
33.5.3.2 What Kind of Triggers Can Fire 29
33.5.3.3 Firing Order 30
33.5.3.4 Crossedition Trigger Execution 31
33.5.4 Creating a Crossedition Trigger 32
33.5.4.1 Coding the Forward Crossedition Trigger Body 33
33.5.5 Transforming Data from Pre- to Post-Upgrade Representation 35
33.5.5.1 Preventing Lost Updates 36
33.5.6 Dropping the Crossedition Triggers 37
33.6 Displaying Information About EBR Features 38
33.6.1 Displaying Information About Editions 38
33.6.2 Displaying Information About Editioning Views 39
33.6.3 Displaying Information About Crossedition Triggers 40
33.7 Using EBR to Upgrade an Application 40
33.7.1 Preparing Your Application to Use Editioning Views 41
33.7.2 Procedure for EBR Using Only Editions 42
33.7.3 Procedure for EBR Using Editioning Views 44
33.7.4 Procedure for EBR Using Crossedition Triggers 45
33.7.5 Rolling Back the Application Upgrade 46
33.7.6 Reclaiming Space Occupied by Unused Table Columns 46
33.7.7 Example: Using EBR to Upgrade an Application 46
33.7.7.1 Existing Application 47
33.7.7.2 Preparing the Application to Use Editioning Views 48
33.7.7.3 Using EBR to Upgrade the Example Application 49

Copyright © 1999, 2025, Oracle and/or its affiliates.

34 Using Transaction Guard

34.1 Problem That Transaction Guard Solves 1
34.2 Solution That Transaction Guard Provides 2
34.3 Transaction Guard Concepts and Scope 3
34.3.1 Logical Transaction Identifier (LTXID) 3
34.3.2 At-Most-Once Execution 4
34.3.3 Transaction Guard Coverage 4
34.3.4 Transaction Guard with XA Transactions 5
34.3.5 Transaction Guard Exclusions 6
34.4 Database Configuration for Transaction Guard 6
34.4.1 Configuration Checklist 7
34.4.2 Transaction History Table 7
34.4.3 Service Parameters 8
34.4.3.1 Example: Adding and Modifying a Service for a Server Pool 8
34.4.3.2 Example: Adding an Administrator-Managed Service 9
34.4.3.3 Example: Modifying a Service (PL/SQL) 9
34.5 Developing Applications That Use Transaction Guard 9
34.5.1 Typical Transaction Guard Usage 10
34.5.2 Details for Using the LTXID 10
34.5.3 Transaction Guard and Transparent Application Failover 11
34.5.4 Using Transaction Guard with ODP.NET 12
34.5.5 Connection-Pool LTXID Usage 12
34.5.6 Improved Commit Outcome for XA One Phase Optimizations 12
34.5.7 Additional Requirements for Transaction Guard Development 13
34.6 Transaction Guard and Its Relationship to Application Continuity 14
34.7 Transaction Guard Support during DBMS_ROLLING Operations 14
34.7.1 Rolling Upgrade Using Transient Logical Standby 14
34.7.2 Transaction Guard Support During Major Database Version Upgrades 15

35 Table DDL Change Notification

35.1 Overview of Table DDL Change Notification 1
35.2 Table DDL Change Notification Terminology 1
35.3 Benefits of Table DDL Change Notification 2
35.4 Features of Table DDL Change Notification 2
35.5 Using Table DDL Change Notification 3
35.6 Registering for Table DDL Change Notification 4
35.6.1 Table-level Registration 5
35.6.2 Schema-level Registration 5
35.7 Unregistering for Table DDL Change Notifications 5
35.8 Supported DDL Events and Commands 6

Copyright © 1999, 2025, Oracle and/or its affiliates.

Index

Copyright © 1999, 2025, Oracle and/or its affiliates.

List of Tables

3-1	Table Annotation Result Cache Modes	15
3-2	Effective Result Cache Table Mode	16
3-3	Client Configuration Parameters (Optional)	21
3-4	Setting Client Result Cache and Server Result Cache	24
3-5	Session Purity and Connection Class Defaults	37
8-1	COMMIT Statement Options	6
8-2	Examples of Concurrency Under Explicit Locking	19
8-3	Ways to Display Locking Information	27
8-4	ANSI/ISO SQL Isolation Levels and Possible Transaction Interactions	29
8-5	ANSI/ISO SQL Isolation Levels Provided by Oracle Database	29
8-6	Comparison of READ COMMITTED and SERIALIZABLE Transactions	34
8-7	Possible Transaction Outcomes	38
8-8	Object Types Supported for CREATE, ALTER, and DROP Commands	47
9-1	SQL Character Data Types	6
9-2	Range and Precision of Floating-Point Data Types	8
9-3	Binary Floating-Point Format Components	9
9-4	Summary of Binary Format Storage Parameters	9
9-5	Special Values for Native Floating-Point Formats	10
9-6	Values Resulting from Exceptions	12
9-7	SQL Datetime Data Types	13
9-8	SQL Conversion Functions for Datetime Data Types	18
9-9	Large Objects (LOBs)	20
9-10	Display Types of SQL Functions	26
9-11	SQL Data Type Families	27
10-1	Identifier Built-in Domains	49
10-2	Tech Built-in Domains	61
10-3	Numeric Built-in Domains	73
10-4	Miscellaneous Built-in Domains	78
11-1	Oracle SQL Pattern-Matching Condition and Functions	2
11-2	Oracle SQL Pattern-Matching Options for Condition and Functions	3
11-3	POSIX Operators in Oracle SQL Regular Expressions	5
11-4	POSIX Operators and Multilingual Operator Relationships	8
11-5	PERL-Influenced Operators in Oracle SQL Regular Expressions	9
11-6	Explanation of the Regular Expression Elements	11
11-7	Explanation of the Regular Expression Elements	12
15-1	Identifier Types that PL/Scope Collects	5

Copyright © 1999, 2025, Oracle and/or its affiliates.

15-2	Usages that PL/Scope Reports	6
16-1	Raw Profiler Output File Indicators	5
16-2	Function Names of Operations that the PL/SQL Hierarchical Profiler Tracks	7
16-3	PL/SQL Hierarchical Profiler Database Tables	8
16-4	DBMSHP_RUNS Table Columns	10
16-5	DBMSHP_FUNCTION_INFO Table Columns	10
16-6	DBMSHP_PARENT_CHILD_INFO Table Columns	11
17-1	DBMSPCC_RUNS Table Columns	3
17-2	DBMSPCC_UNITS Table Columns	3
17-3	DBMSPCC_BLOCKS Table Columns	3
18-1	Commonly Used Packages in the PL/SQL Web Toolkit	3
18-2	Mapping Between mod_plsql and Embedded PL/SQL Gateway DAD Attributes	7
18-3	Mapping Between mod_plsql and Embedded PL/SQL Gateway Global Attributes	9
18-4	Authentication Possibilities for a DAD	12
19-1	Continuous Query Notification Registration Options	12
19-2	Attributes of CQ_NOTIFICATION$_REG_INFO	22
19-3	Quality-of-Service Flags	24
19-4	Attributes of CQ_NOTIFICATION$_DESCRIPTOR	48
19-5	Attributes of CQ_NOTIFICATION$_TABLE	49
19-6	Attributes of CQ_NOTIFICATION$_QUERY	50
19-7	Attributes of CQ_NOTIFICATION$_ROW	50
20-1	PL/SQL Packages and Their Java and JavaScript Equivalents	29
21-1	Parameter Data Type Mappings	17
21-2	External Data Type Mappings	19
21-3	Properties and Data Types	23
22-1	Oracle Flashback Version Query Row Data Pseudocolumns	11
22-2	Flashback TRANSACTION_BACKOUT Options	19
22-3	Static Data Dictionary Views for Flashback Archive Files	27
25-1	General Structure of EPC Encodings	2
25-2	Identity Code Package ADTs	18
25-3	MGD_ID ADT Subprograms	18
25-4	DBMS_MGD_ID_UTL Package Utility Subprograms	19
25-5	Definition and Description of the MGD_ID_CATEGORY Metadata View	21
25-6	Definition and Description of the USER_MGD_ID_CATEGORY Metadata View	21
25-7	Definition and Description of the MGD_ID_SCHEME Metadata View	21
25-8	Definition and Description of the USER_MGD_ID_SCHEME Metadata View	21
28-1	LRA and Saga Annotations	15

Copyright © 1999, 2025, Oracle and/or its affiliates.

28-2	saga_finalization$ table entries	42
29-1	Reservation Table Columns	6
30-1	Required XA Features Published by Oracle Database	5
30-2	XA Library Subprograms	6
30-3	Oracle XA Interface Extensions	6
30-4	Required Fields of xa_open string	9
30-5	Optional Fields in the xa_open String	10
30-6	TX Interface Functions	14
30-7	TPM Replacement Statements	16
30-8	Sample Trace File Contents	21
30-9	Tightly and Loosely Coupled Transaction Branches	24
31-1	Errors Summary	48
32-1	Database Object Status	4
32-2	Operations that Cause Fine-Grained Invalidation	5
32-3	Data Type Classes	17
33-1	*_ Dictionary Views with Edition Information	38
33-2	*_ Dictionary Views with Editioning View Information	39
34-1	LTXID Condition or Situation, Application Actions, and Next LTXID to Use	11
34-2	Transaction Manager Conditions/ Situations and Actions	13

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

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 relevant to your job. To use this guide most effectively, you also need a working knowledge of:

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

Related Documents

For more information, see these documents in the Oracle Database documentation set:

- *Oracle Database PL/SQL Language Reference*
- *Oracle Call Interface Programmer's Guide*
- *Oracle AI Database JSON Developer’s Guide*
- *Oracle AI Database SODA for PL/SQL Developer'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*

Copyright © 1999, 2025, Oracle and/or its affiliates.

- *Oracle Database Sample Schemas*

> **See Also**
>
> - https://www.oracle.com/database/technologies/application-development.html

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.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

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*.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Changes in This Release for Oracle AI Database Development Guide

This is a summary of important changes in Oracle AI Database 26ai for *Oracle AI Database Development Guide*.

- New Features in 26ai
- Deprecated Features

New Features in 26ai

The following are the new features in *Oracle Database Development Guide* for Oracle AI Database, Release 26ai.

Blockchain Table Log History

This feature allows changes to one or more regular user tables to be tracked in a blockchain table, which is maintained by Oracle Database as part of the Flashback Data Archive.

See Protecting Flashback Archive Data.

Data Use Case Domains

A data use case domain is a dictionary object that belongs to a schema and encapsulates a set of optional properties and constraints for common values, such as credit card numbers or email addresses. After you define a data use case domain, you can define table columns to be associated with that domain, thereby explicitly applying the domain's optional properties and constraints to those columns.

See Data Use Case Domains.

Flashback Time Travel Enhancements

Flashback Time Travel can automatically track and archive transactional changes to tables. Flashback Time Travel creates archives of the changes made to the rows of a table and stores the changes in history tables. It also maintains a history of the evolution of the table's schema. By maintaining the history of the transactional changes to a table and its schema, Flashback Time Travel enables you to perform operations, such as Flashback Query `AS OF` and `VERSIONS`), on the table to view the history of the changes made during transaction time.

See Using Flashback Time Travel.

Copyright © 1999, 2025, Oracle and/or its affiliates.

IF [NOT] EXISTS Syntax Support

DDL object creation, modification, and deletion now support the `IF EXISTS` and `IF NOT EXISTS` syntax modifiers. This enables you to control whether an error should be raised if a given object exists or does not exist.

The `IF [NOT] EXISTS` syntax can simplify error handling in scripts and by applications.

See Using `IF EXISTS` and `IF NOT EXISTS`.

Implicit Connection Pooling for Database Resident Connection Pooling

This feature enables the automatic assignment of Database Resident Connection Pooling (DRCP) servers to and from an application connection at runtime when the application starts and finishes database operations, even if the application does not explicitly close the connection.

This feature can provide better scalability and efficient usage of database resources for applications that do not use application connection pooling.

See Implicit Connection Pooling.

Lock-Free Reservation

Lock-free Reservation enables concurrent transactions to proceed without being blocked on updates of heavily updated rows. Lock-free reservations are held on the rows instead of locking them. Lock-free Reservation verifies if the updates can succeed and defers the updates until the transaction commit time.

Lock-free Reservation improves the end user experience and concurrency in transactions.

See Lock-Free Reservation.

Multilingual Engine JavaScript Modules and Environments

Multilingual Engine (MLE) Modules and Environments allow JavaScript code to persist and be managed in the database. Call specifications provide a means to call JavaScript functions from an MLE module, anywhere that you can call PL/SQL functions.

The introduction of JavaScript Modules and Environments as schema objects in Oracle Database allows developers to follow established and well-known workflows used in client-side JavaScript development. Complex projects can be broken down into smaller, more manageable pieces that can be worked on independently by team members.

See Overview of JavaScript.

Multiple Named Pools for DRCP

Database Resident Connection Pooling (DRCP) now supports multiple named pools. The `DBMS_CONNECTION_POOL.ADD_POOL()` and `DBMS_CONNECTION_POOL.REMOVE_POOL()` procedures are added to the `DBMS_CONNECTION_POOL` package. Oracle Net connection string syntax is enhanced, so you can specify a pool name for each connection. Existing procedures can be used to start, stop, or configure named pools.

Having multiple pools allows finer control on the DRCP pool usage. It helps prevent situations where some applications dominate the use of a single pool.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Using Multi-pool DRCP.

Precheckable Constraints using JSON Schema

A check constraint can be checked outside the database. For this, you mark the check constraint with the PRECHECK keyword. You can create a JSON Schema document from a precheckable check constraint. This means that data can be checked for validity outside of the database using an external JSON Schema validator.

A JSON schema can describe rules that define valid data (rules that correspond to the check constraint). This lets you use the database as a central repository of business rules.

A JSON schema can be used to check data (in application code, for instance) prior to sending it to the database, which can provide earlier detection of invalid data. This provides more resilient applications and reduces potential system downtime and the need to clean up 'bad' data.

See Using PRECHECK to Pre-validate a CHECK Constraint.

Reset Database Session State

The reset session state feature clears the session state set by the application when the request ends. The RESET_STATE database service attribute cleans up dirty sessions so that the applications cannot see the state of these sessions. This feature applies to all applications that connect to the database using database services.

The RESET_STATE feature enables you to clean the session state at the end of each request so that the database developers do not have to clean the session state manually. By using this feature, you can ensure that there are no data leaks from a previous session.

See Reset Database Session State to Prevent Application State Leaks, Use RESET_STATE.

Saga APIs Using Oracle Saga Framework

Oracle Saga APIs are implemented in the database and provide a framework to implement transactional semantics for microservices built with Oracle Database. The orchestrator Saga framework provides a way to maintain atomic data consistency across microservices.

Sagas are concurrent and execute local transactions in each participant database making it more efficient than distributed ACID transactions, thereby simplifying application code and increasing developer productivity.

See Developing Applications with Sagas.

Schema Annotations

Schema annotations enable you to store and retrieve metadata about database objects. These are name-value pairs or simply a name. These are freeform text fields applications can use to customize business logic or user interfaces.

Schema annotations help you use database objects in the same way across all applications. This simplifies development and improves data quality.

See Schema Annotations.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Sessionless Transactions

In Sessionless Transactions, after you start a transaction, you have the flexibility to suspend and resume the transaction during its lifecycle. The session or connection can be released back to the pool, allowing it to be reused by other transactions, and therefore effectively multiplexing transactions across sessions or connections.

See Developing Applications with Sessionless Transactions

Shut Down Connection Draining for DRCP

A new, optional `DRAINTIME` argument to `DBMS_CONNECTION_POOL.STOP_POOL()` allows active DRCP pools to be closed after a specified connection drain time, or be closed immediately without waiting for connections to be idle. This feature gives DBAs better control over DRCP usage and configuration.

See Shut Down Connection Draining for DRCP.

Table DDL Change Notification

Applications can now be notified when DDL changes occur on tables.

Applications that need or want to be aware of table metadata can now be notified of DDL changes rather than having to continuously poll for them.

See Table DDL Change Notification.

Deprecated Features

This section lists the deprecated features in Oracle AI Database, Release 26ai for *Database Development Guide*.

Oracle recommends that you do not use deprecated features/values in new applications. Support for deprecated features is for backward compatibility only.

Deprecation of Traditional Audit Initialization Parameters

With the desupport of creating and modifying traditional audit policies with Oracle Database 23, the traditional audit parameters are deprecated.
Traditional audit policies are available after upgrading, but the initialization parameters associated with those policies are deprecated. These deprecated parameters include the following:

- `AUDIT_TRAIL`
- `AUDIT_SYS_OPERATIONS`
- `AUDIT_FILE_DEST`
- `AUDIT_SYSLOG_LEVEL`

Oracle recommends that you migrate to unified auditing as soon as possible, because these traditional audit parameters can be removed in a future database release.

Deprecation of Oracle JDBC-OCI (Thick) Driver

The Oracle JDBC Oracle Call Interface (OCI) Type 2 client driver (also known as a "thick" driver) is deprecated in Oracle AI Database 26ai.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Most Java applications that use Open Database Connectivity (ODBC) with Oracle JDBC drivers use the Thin driver. To enable Oracle to allocate resources to better address customer requirements, Oracle is deprecating the JDBC-OCI driver.

Deprecation of SQLJ

Starting with Oracle AI Database 26ai, the SQLJ method of embedding SQL statements in Java code is deprecated.
In place of SQLJ, Oracle recommends that you directly use the Java Database Connectivity (JDBC) APIs (dynamic SQL, prepared statements or PL/SQL blocks).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

Related links

- *Oracle AI Database Get Started with Oracle AI Database Development* for more information about Database concepts and techniques
- `http://asktom.oracle.com` for books and articles about Oracle database concepts by Tom Kyte

Chapters:

- Design Basics
- Connection Strategies for Database Applications
- Performance and Scalability
- Designing Applications for Oracle Real-World Performance
- Security
- High Availability
- Advanced PL/SQL Features

Copyright © 1999, 2025, Oracle and/or its affiliates.

1

Design Basics

This chapter explains several important design goals for database developers.

This chapter contains the following 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

See Also

- Database Development Fundamentals for high-level database concepts
- *Oracle AI Database Get Started with Oracle AI Database Development* for more information about Oracle Database concepts and techniques

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.

See Also

- Performance and Scalability
- *Oracle AI Database Performance Tuning Guide*
- *Oracle AI Database SQL Tuning Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

See Also

- Performance and Scalability
- *Database 2 Day Developer's Guide*

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

Related Topics

- *Oracle AI Database Data Cartridge Developer's Guide*

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.

Related Topics

- Developing Applications with Multiple Programming Languages

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

- Invoking Stored PL/SQL Functions from SQL Statements 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 AI 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

See Also

- *Oracle AI Database Security Guide* for more information about security.
- Security for information about considerations and techniques for providing security.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - High Availability for information about important considerations and techniques for providing high availability.

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 AI 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 AI 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

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

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 primary 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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
- Partitioning for more information about how to partition materialized views like tables.

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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

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, 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 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.

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**
>
> Creating and using a table with valid time support and changing data using Temporal Validity Support assume that the user has privileges to create tables and perform data manipulation language (DML) and (data definition language) DDL operations on them.

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

The following example creates a table with Temporal Validity Support, inserts rows, and issues queries whose results depend on the valid start date and end date for individual rows.

```
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');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
-- 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 did not create the `START_TIME` and `END_TIME` columns, you could use the following statement to create the same:

```
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 the 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
-- 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;
```

Related Links

- `CREATE TABLE`, `ALTER TABLE` , and `SELECT`
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database VLDB and Partitioning Guide*
- Using Oracle Flashback Technology
- Multitenant Container Database Restrictions for Oracle Flashback Technology

Copyright © 1999, 2025, Oracle and/or its affiliates.

2

Connection Strategies for Database Applications

A **database connection** is a physical communication pathway between a client process and a database instance. A **database session** is a logical entity in the database instance memory that represents the state of a current user login to a database. A session lasts from the time the user is authenticated by the database until the time the user disconnects or exits the database application. A single connection can have 0, 1, or more sessions established on it.

Most OLTP performance problems that the Oracle Real-World Performance group investigates relate to the application connection strategy. For this reason, designing a sound connection strategy is crucial for application development, especially in enterprise environments that must scale to meet increasing demand.

Topics:

- Design Guidelines for Connection Pools
- Design Guideline for Login Strategy
- Design Guideline for Preventing Programmatic Session Leaks
- Using Runtime Connection Load Balancing

2.1 Design Guidelines for Connection Pools

A **connection pool** is a cache of connections to an Oracle database that is usable by an application.

At run time, the application requests a connection from the pool. If the pool contains a connection that can satisfy the request, then it returns the connection to the application. The application uses the connection to perform work on the database, and then returns the connection to the pool. The released connection is then available for the next connection request.

In a **static connection pool**, the pool contains a fixed number of connections and cannot create more to meet demand. Thus, if the pool cannot find an idle connection to satisfy a new application request, then the request is queued or an error is returned. In a **dynamic connection pool**, however, the pool creates a new connection, and then returns it to the application. In theory, dynamic connection pools enable the number of connections in the pool to increase and decrease, thereby conserving system resources that are otherwise lost on maintaining unnecessary connections. However, in practice, a dynamic connection pool strategy allows for potential connection storms and over-subscription problems.

2.1.1 Connection Storms

A **connection storm** is a race condition in which application servers initiate an increasing number of connection requests, but the database server CPU is unable to schedule them immediately, which causes the application servers to create more connections.

During a connection storm, the number of database connections can soar from hundreds to thousands in less than a minute.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Dynamic connection pools are particularly prone to connection storms. As the number of connection requests increases, the database server becomes oversubscribed relative to the number of CPU cores. At any given time, only one process can run on a CPU core. Thus, if 32 cores exist on the server, then only 32 processes can be doing work at one time. If the application server creates hundreds or thousands of connections, then the CPU becomes busy trying to keep up with the number of processes fighting for time on the system.

Inside the database, wait activity increases as the number of active sessions increase. You can observe this activity by looking at the wait events in ASH and AWR reports. Typical wait events include latches on enqueues, row cache objects, latch free, enq: TX - index contention, and buffer busy waits. As the wait events increase, the transaction throughput decreases because sessions are unable to perform work. Because the server computer is oversubscribed, monitoring tool processes must fight for time on the CPU. In the most extreme case, using the keyboard becomes impossible, making debugging difficult.

Video

RWP #13: Large Dynamic Connection Pools - Part 1

2.1.2 Guideline for Preventing Connection Storms: Use Static Pools

The Oracle Real-World Performance group recommends that you use static connection pools rather than dynamic connection pools.

Over years of diagnosing connection storms, the Oracle Real-World Performance group has discovered that dynamic connection pools often use too many processes for the necessary workload. A prevalent myth is that a dynamic connection pool creates connections as required and reduces them when they are not needed. In reality, when the connection pool is exhausted, application servers enable the size of the pool of database connections to increase rapidly. The number of sessions increases with little load on the system, leading to a performance problem when all the sessions become active.

Because dynamic connection pools can destabilize the system quickly, the Oracle Real-World Performance group recommends that you use static rather than dynamic connection pools.

Reducing the number of connections reduces the stress on the CPU, which leads to faster response time and higher throughput. This result may seem paradoxical. Performance improves for the following reasons:

- Fewer sessions means fewer contention-related wait events inside the database. After reducing connections, the CPU that formerly consumed cycles on latches and arbitrating contention can spend more time processing database transactions.
- As the number of connections decreases, connections can stay scheduled on the CPU for longer. As a result, all memory pages associated with these processes stay resident in the CPU cache. They become increasingly efficient at scheduling, and stall less in memory

As a rule of thumb, the Oracle Real-World Performance group recommends a 90/10 ratio of %user to %system CPU utilization, and an average of no more than 10 processes per CPU core on the database server. The number of connections should be based on the number of CPU cores and not the number of CPU core threads. For example, suppose a server has 2 CPUs and each CPU has 18 cores. Each CPU core has 2 threads. Based on the Oracle Real-Wold Performance group guidelines, the application can have between 36 and 360 connections to the database instance.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Video**
>
> RWP #14: Large Dynamic Connection Pools - Part 2

2.2 Design Guideline for Login Strategy

A problem facing all database developers is how and when the application logs in to the database to initiate transactions.

In a suboptimal design, the database application performs the following steps for every SQL request:

1. Log in to the database.
2. Issue a SQL request, such as an `INSERT` or `UPDATE` statement.
3. Log out of the database.

Applications that use a login/logout strategy may meet functional requirements. Also, they may perform well when the number of transactions per second is low. However, logging in and out of the database is an extremely resource-intensive operation. The Oracle Real-World Performance group has found that applications that use such algorithms do not scale well and can cause severe performance problems, especially when used with dynamic connection pools. A login/logout strategy usually uses no connection pool.

If an application uses a login/logout design, and if the DBAs and developers do not realize the source of the problem, then the first symptoms may be low database throughput and erratic, excessively high response times. A diagnostic investigation of the database may show that relatively few sessions are active, while resource contention is low.

The clue to the suboptimal performance is that the number of logins per second is close to the number of transactions per second. When a login/loutout per transaction stategy is used, the database instance and operating system perform a lot of work behind the scenes to create the new process, database connection, and associated memory area. Many of these steps are serialized, leading to low CPU utilization combine with low transaction througput.

For the preceding reasons, the Oracle Real-World Performance group strongly recommends against a login/logout design for any application that must scale to support a high number of transactions.

> **Video**
>
> RWP #2 Bad Performance with Cursors and Logons

2.3 Design Guideline for Preventing Programmatic Session Leaks

A **session leak** occurs when a program loses a connection, but its session remains active in the database instance. A leaked session is programmatically lost to the application.

An optimally designed application prevents session leaks. Typically, session leaks occur because of exceptions caught by the application. If the application does not handle the

Copyright © 1999, 2025, Oracle and/or its affiliates.

exception correctly, then it may terminate the connection without executing a commit or rollback, thus leaking the session.

Session leaks can cause severe problems for database performance and data integrity. Typically, the problems take the following forms:

- Drained connection pools
- Lock leaks
- Logical corruption

2.3.1 Drained Connection Pools

Design flaws can cause connection pools to drain.

For example, assume that an application design flaw causes it to leak sessions consistently. Even if the leak rate is low, a dynamic connection pool leads to an ever-increasing number of sessions become programmatically impossible to use.

The effect is to reduce the usable connection pool and prevent the remaining connections from keeping up with the workload. The number of unusable sessions climbs until there are no usable connections left in the pool.

2.3.2 Checking for Session Leaks

Session leaks occur due to issues in the application or application server and cannot be fixed from the database alone. The issue needs to be addressed in the application or application server. An easy way to check for session leaks is by modifying the connection pool to use one connection to the database and test the application. Testing with one connection makes it is easier to find the root cause of the problem in the application.

2.3.3 Lock Leaks

A **lock leak** is typically a side-effect of a session leak.

For example, a leaked session that was in the middle of a batch update may hold TX locks on multiple rows in a table. If a leaked session is holding a lock, then sessions that want to acquire the lock form a queue behind the leaked session.

The program that is holding the lock is waiting for interaction from the client to release that lock, but because the connection is lost programmatically, the message will not be sent. Consequently, the database cannot commit or roll back any transaction active in the session.

2.3.4 Logical Corruption

A leaked session can contain uncommitted changes to the database. For example, a transaction is partway through its work when the database connection is unexpectedly released.

This situation can lead to the following problems:

- The application reports an error to the UI. In this case, customers may complain that they have lost work, for example, a business order or a flight schedule
- The UI receive a commit message even though no commit or rollback has occurred. This is the worst case, because a subsequent transaction might commit both its own work and the

Copyright © 1999, 2025, Oracle and/or its affiliates.

half of the transaction from the leaked session. In this case, the database is logically corrupted.

2.4 Using Runtime Connection Load Balancing

Topics:

- About Runtime Connection Load Balancing
- Enabling and Disabling Runtime Connection Load Balancing
- Receiving Load Balancing Advisory FAN Events

2.4.1 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.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- `cdemosp.c` in the directory `demo`
- Using Database Resident Connection Pool
- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Universal Connection Pool for JDBC Java API Reference*
- *Oracle Data Provider for .NET Developer's Guide*

2.4.2 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

2.4.2.1 OCI

For an OCI client application, runtime connection load balancing is enabled by default in an Oracle Database 11*g* Release 1 (11.1) or later client communicating with a server of Oracle Database 10*g* 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()`

Copyright © 1999, 2025, Oracle and/or its affiliates.

2.4.2.2 OCCI

For an OCCI client application, runtime connection load balancing is enabled by default in an Oracle Database 11*g* Release 1 (11.1) or later client communicating with a server of Oracle Database 10*g* 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

2.4.2.3 JDBC

In the JDBC environment, runtime connection load balancing is enabled by default in an Oracle Database 11*g* Release 1 (11.1) or later client communicating with a server of Oracle Database 10*g* 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

2.4.2.4 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 and make sure "`Pooling=true`" (default).

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

2.4.3 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

3

Performance and Scalability

Explains techniques for designing performance and scalability into the database and database applications.

Topics:

- Performance Strategies
- Tools for Performance
- Monitoring Database Performance
- Testing for Performance
- 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
- Memoptimize Pool
- Oracle RAC Sharding

3.1 Performance Strategies

Topics:

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

3.1.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.1.1.1 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.

3.1.1.2 Create the Database Design for the Application

To create the database design, you must first create the logical design, and then translate this logical design into a physical design.

Topics:

- Create the Logical Design
- Create the Physical Design

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

3.1.1.2.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

3.1.1.2.2 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.

3.1.1.3 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.

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.

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

4. Implement the application in the production environment.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle AI Database SQL Tuning Guide* for more information about SQL tuning
- *Oracle AI Database Get Started with Performance Tuning* for more information about SQL tuning
- *Oracle AI Database Performance Tuning Guide* for more information about workload testing, modeling, and implementation
- Tools for Performance for information about tools for testing performance
- Benchmarking Your Application for information about benchmarking your application
- *Oracle Database Performance Tuning Guide* for more information about deploying new applications

3.1.1.4 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.

3.1.2 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.

3.1.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

3.2 Tools for Performance

Several tools report runtime performance information about your application.

Topics:

- 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

3.2.1 DBMS_APPLICATION_INFO Package

Use the `DBMS_APPLICATION_INFO` package with the SQL Trace facility 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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `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
- SQL Trace Facility (SQL_TRACE) for more information about `DBMS_APPLICATION_INFO` package with the SQL Trace facility
- *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

3.2.2 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.

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

- EXPLAIN PLAN Statement
- *Oracle Database SQL Tuning Guide* for more information about `SQL_TRACE` and `TKPROF`

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.2.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

3.3 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

3.3.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.3.2 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

3.3.3 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

3.3.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *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

3.4 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.
- 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 AI Database Performance Tuning Guide* for more information about performance tuning, workload testing, modeling, and implementation
- Automatic Database Diagnostic Monitor (ADDM)and SQL Advisors and Memory Advisors for more information about ADDM and SQL Tuning Advisor respectively
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about tuning SQL statements using the SQL Tuning Advisor

3.5 Using Client Result Cache

Topics:

- About Client Result Cache
- Benefits of Client Result Cache
- Guidelines for Using Client Result Cache
- Setting Result Cache Integrity
- 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

3.5.1 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 the 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

Note

Oracle Database 18.1 onwards, Client Result Cache supports dynamic binding

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
- *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*
- SQL hints and `RESULT_CACHE` for clauses of `ALTER TABLE` and `CREATE TABLE`
- `RESULT_CACHE_MODE`
- Setting Result Cache Integrity
- `CLIENT_RESULT_CACHE_STAT$` view

3.5.2 Benefits of Client Result Cache

The benefits of the client result cache are:

- Client result cache is on the client, therefore 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

  - 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**
>
> `OCIStmtExecute()` and `OCIStmtFetch2()` in *Oracle Call Interface Programmer's Guide*

3.5.3 Guidelines for Using Client Result Cache

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

1. Query

   For a specific query, you can enable or disable the client result cache with a SQL hint. 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 the client result cache with a table annotation, without changing the application.

3. Session

   For all queries in your database session, you can enable or disable the client result cache with a session parameter, without changing the application.

Higher-level settings take precedence over lower-level ones.

Oracle recommends enabling the client result cache only for queries on read-only and seldom-updated tables (tables that are rarely updated). That is, enable the 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 the 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 the 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 `OCIStmtExecute()` call of every OCI statement handle call always goes to the server even if there might be a valid cached result set. An `OCIStmtExecute()` call must be made for each statement handle to be able to match a cached result set. Oracle recommends

Copyright © 1999, 2025, Oracle and/or its affiliates.

that applications either have their own statement caching for OCI statement handles or use OCI statement caching so that `OCIStmtPrepare2()` 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 `OCIStmtExecute()` or `OCIStmtFetch2()` calls that transparently create this cached result set must fetch rows until ORA-01403 (No Data Found) is returned. Subsequent `OCIStmtExecute()` or `OCIStmtFetch2()` calls that match a locally cached result set need not fetch to completion.

See Also

- SQL Hints
- Table Annotation
- Session Parameter
- Result Cache Mode Use Cases
- `OCIStmtExecute()`
- `OCIStmtPrepare2()`
- `OCIStmtFetch2()`

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

3.5.3.1 SQL Hints

The SQL hint `RESULT_CACHE` or `NO_RESULT_CACHE` applies to a single query, for which it enables or disables the 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 `OCIStmtPrepare()` and `OCIStmtPrepare2()` 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database SQL Language Reference* for general information about SQL hints
- `OCIStmtPrepare()`, `OCIStmtPrepare2()` in *Oracle Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide* for information about SQL hints in JDBC

3.5.3.2 Table Annotation

A **table annotation** enables or disables the 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 the client result cache, specify `RESULT_CACHE (MODE FORCE)`; to disable it, use `RESULT_CACHE (MODE DEFAULT)`.

Table 3-1 summarizes the table annotation result cache modes.

Table 3-1 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 `NO_RESULT_CACHE` hint is specified for the query. If one or more tables named in the query are set to `DEFAULT`, then the effective table annotation for that query is `DEFAULT`.

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

3.5.3.3 Session Parameter

The session parameter `RESULT_CACHE_MODE` enables or disables the 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 the client result cache, set `RESULT_CACHE_MODE` to `FORCE`; to disable it, set `RESULT_CACHE_MODE` to `MANUAL`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

3.5.3.4 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 3-2 shows.

Table 3-2 Effective Result Cache Table Mode

Table Annotation Result Cache Mode	RESULT_CACHE_MODE = MANUAL	RESULT_CACHE_MODE = FORCE
`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.

3.5.3.5 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.5.3.6 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.

3.5.3.7 Queries Never Result Cached in Client Result Cache

Results of the following queries are never cached in the 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*.

3.5.4 Setting Result Cache Integrity

With Oracle AI Database 26ai, you can control the integrity of the result cache using the database initialization parameter: `RESULT_CACHE_INTEGRITY`. The `RESULT_CACHE_INTEGRITY` parameter enables you to specify whether the result cache must consider caching the queries that use potentially non-deterministic constructs, such as PL/SQL functions that are not declared as deterministic. In other words, you can use the `RESULT_CACHE_INTEGRITY` parameter to ensure that only explicitly deterministic queries are cached.

Enforcing the need to explicitly declare objects as deterministic before being considered for result caching can eventually improve the code quality while also rule out the chance of accidentally caching objects that should not be cached.

Copyright © 1999, 2025, Oracle and/or its affiliates.

You can set the RESULT_CACHE_INTEGRITY parameter as ENFORCED or TRUSTED, which directs the database to do one of the following:

- When RESULT_CACHE_INTEGRITY = ENFORCED, enforce result cache integrity regardless of the setting of the RESULT_CACHE_MODE initialization parameter or any specified hints, allowing only deterministic constructs to be eligible for result caching. For example, queries using PL/SQL functions that are not explicitly declared as deterministic are not cached.
- When RESULT_CACHE_INTEGRITY = TRUSTED, honor the setting of the RESULT_CACHE_MODE initialization parameter and any specified hints, and consider queries that use possibly non-deterministic constructs as candidates for result caching. For example, queries using PL/SQL functions that are not explicitly declared as deterministic may be cached. Results that are certain to be non-deterministic (for example, SYSDATE or constructs involving SYSDATE) are not cached.

The RESULT_CACHE_INTEGRITY parameter defaults to TRUSTED for Oracle AI Database 26ai (unmanaged) and ENFORCED for Oracle Autonomous AI Database 26ai.

Note

Setting the RESULT_CACHE_INTEGRITY parameter to TRUSTED achieves backward compatibility with databases prior to 26ai.

If the RESULT_CACHE_MODE parameter is set to FORCE or FORCE_TEMP, then Oracle recommends setting the RESULT_CACHE_INTEGRITY parameter to ENFORCED, which ensures that only deterministic constructs are eligible for result caching. This prevents the caching of non-deterministic constructs, which can potentially cause material changes to results.

See Also

- RESULT_CACHE_INTEGRITY and RESULT_CACHE_MODE in *Database Reference*
- Setting the Result Cache Mode in *Database Performance Tuning Guide*
- SQL hints and RESULT_CACHE in *SQL Language Reference* for clauses of ALTER TABLE and CREATE TABLE

3.5.5 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 invalidations 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 the client result cache.

This consistency mechanism ensures that the 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 the 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, the 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 the 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*

3.5.6 Deployment-Time Settings for Client Result Cache

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

- Server initialization parameters
- Client configuration parameters
- Table annotations
- `RESULT_CACHE_MODE` session parameter

See Also

- Server Initialization Parameters
- Client Configuration Parameters
- Table Annotation
- Session Parameter
- Client-Side Deployment Parameters

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.5.6.1 Server Initialization Parameters

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

- COMPATIBLE
- CLIENT_RESULT_CACHE_SIZE
- CLIENT_RESULT_CACHE_LAG

3.5.6.1.1 COMPATIBLE

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

3.5.6.1.2 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 the client result cache is disabled. To enable the client result cache, set `CLIENT_RESULT_CACHE_SIZE` to at least 32768 bytes (32 kilobytes (KB)).

If the 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 the client result cache is disabled on the server, then `OCI_RESULT_CACHE_MAX_SIZE` is ignored and the client result cache cannot be enabled.

Oracle recommends either enabling the client result cache for all Oracle Real Application Clusters (Oracle RAC) nodes or disabling the 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
- ((Possible number of result sets to be cached) * (average size of a row in a result set) * (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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.5.6.1.3 CLIENT_RESULT_CACHE_LAG

Specifies the maximum time in milliseconds that the 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.

3.5.6.2 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, 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 3-3 describes the equivalent `oraaccess.xml` and `sqlnet.ora` client configuration parameters.

Table 3-3 Client Configuration Parameters (Optional)

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

Related Topics

- *Oracle Call Interface Developer's Guide*

3.5.7 Client Result Cache Statistics

On round trips to the server from the OCI client, OCI periodically sends the 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

cached result sets invalidated. The number of cache misses for queries is at least equal to the number of Create Counts in the 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

3.5.8 Validation of Client Result Cache

Some ways to validate client result cache are:

- Measure Execution Times
- Query V$MYSTAT
- Query V$SQLAREA

3.5.8.1 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.

3.5.8.2 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

3.5.8.3 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 `step 3` and then run the query in step `step 1` a few times.

3.5.9 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 the server result cache caches result sets and query fragments in server SGA memory.

You can enable the client result cache independently of the server result cache, though they share the result cache SQL hints, table annotations, and session parameter `RESULT_CACHE_MODE`. Table 3-4 shows the result cache association for result cache parameters, the PL/SQL package `DBMS_RESULT_CACHE`, and result cache views.

> **See Also**
>
> *Oracle Database Concepts*

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 3-4 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_SIZE` • `CLIENT_RESULT_CACHE_LAG`	client result cache
SQL hints: • `RESULT_CACHE` • `NO_RESULT_CACHE`	client result cache, server result cache
`sqlnet.ora OCI_RESULT_CACHE*` parameters: • `OCI_RESULT_CACHE_MAX_SIZE` • `OCI_RESULT_CACHE_MAX_RSET_SIZE` • `OCI_RESULT_CACHE_MAX_RSET_ROWS`	client result cache
`CLIENT_RESULT_CACHE_STATS$` view	client result cache
`RESULT_CACHE_MODE` parameter	client result cache, server result cache
All other `RESULT_CACHE_*` parameters (for example, `RESULT_CACHE_MAX_SIZE`)	server result cache
`DBMS_RESULT_CACHE` package	server result cache
`V$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

3.5.10 Client Result Cache Demo Files

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

3.5.11 Other Supported Features

Caching Data Dictionary Queries

Starting with Oracle Database 23.7, Client Result Cache supports the caching of data dictionary queries. Data dictionary objects are owned by the SYS user. Caching of data dictionary data benefits any application that frequently queries data dictionary objects.

The `sys_context()` and `userenv()` queries are also cached in the client. Because the query result can be session specific, such cached data is private to the session.

Caching JSON and Vector Data

Starting with Oracle Database 23.7, Client Result Cache supports the caching of JSON and vector columns. When JSON and vector columns are cached, the complete data is pre-fetched regardless of the prefetch size.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> When a JSON or vector column is read as, say CLOB, the data is not stored in the cache during the first execution. Therefore, when the same query is re-executed, the result is not read from the cache and the query request is sent to the server or database. Similarly, when the data is read as a JSON or vector type, the re-execution of the same query cannot be done using other data types such as CLOB or BLOB.

3.5.12 Client Result Cache Compatibility with Previous Releases

To use the client result cache, applications must be relinked with Oracle Database 11*g* Release 1 (11.1) or later client libraries and be connected to an Oracle Database 11*g* 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 `OCIStmtPrepare()` and `OCIStmtPrepare2()` calls.

> **See Also**
>
> `OCIStmtPrepare()`, `OCIStmtPrepare2()` in *Oracle Call Interface Programmer's Guide*

3.6 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 more information about using statement caching in ODP.NET applications
- *Oracle AI Database Developer's Guide to the Oracle Precompilers*, *Pro*C/C++ Developer's Guide*, and *Pro*COBOL Developer's Guide* for more information about using dynamic SQL statement caching in precompiler applications that rely on dynamic SQL statements

3.7 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.

See Also

- *Oracle Call Interface Programmer's Guide*

3.8 Client-Side Deployment Parameters

Beginning with Oracle Database 12*c* Release 1 (12.1.0.1), OCI deployment parameters are available in a new configuration file (`oraaccess.xml`).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

3.9 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**
>
> - 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

3.10 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.

- About Database Resident Connection Pool
- Configuring DRCP
- Using Multi-pool DRCP
- Sharing Proxy Sessions

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Using JDBC with DRCP
- Using OCI Session Pool APIs with DRCP
- Session Purity
- Connection Class
- Session Purity and Connection Class Defaults
- Setting the Purity and Connection Class in the Connection String
- Starting Database Resident Connection Pool
- Shut Down Connection Draining for DRCP
- Enabling DRCP
- Connecting to a Pool in Multi-pool DRCP
- Implicit Connection Pooling
- 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
- Using DRCP with Oracle Database Native Network Encryption
- DRCP Restrictions
- Using DRCP with Custom Pools
- Explicitly Marking Sessions Stateful or Stateless
- Using DRCP with Oracle Real Application Clusters
- DRCP with Data Guard

3.10.1 About Database Resident Connection Pool

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

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 configurable at the PDB level, or can be 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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).

See Also

- *Oracle Database Concepts* for details about DRCP architecture
- *Oracle Database Administrator's Guide* for more information about switch service enhancements.

3.10.2 Configuring DRCP

You can configure the DRCP at either the CDB or PDB level.

The database parameter `ENABLE_PER_PDB_DRCP` controls whether DRCP is configured in CDB DRCP mode or per-PDB DRCP mode.

You can configure Database Resident Connection Pool (DRCP) to use a CDB wide pool by setting the database parameter `ENABLE_PER_PDB_DRCP=FALSE`. The database administrator

Copyright © 1999, 2025, Oracle and/or its affiliates.

must start and configure DRCP using the DBMS_CONNECTION_POOL package in the ROOT container. In CDB DRCP mode, you cannot use subprograms in the DBMS_CONNECTION_POOL package to manage the pool at the PDB level. You can monitor pools usage and statistics in the ROOT container pertaining to the CDB by querying the GV$CPOOL_STATS, GV$CPOOL_CC_STATS, GV$SPOOL_CONN_INFO, GV$AUTHPOOL_STATS and GV$SPOOL_CC_INFO views. The SYS user in the PDB can view statistics from GV$SPOOL_CC_INFO and GV$AUTHPOOL_STATS. 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.

Alternatively, the database administrator can enable the per-PDB DRCP mode by setting the database parameter ENABLE_PER_PDB_DRCP=TRUE. In this mode, you can create an isolated set of pooled servers for each pluggable database. The DRCP connection broker is started by the ROOT container. The PDB administrator can configure, manage and monitor independent pools customized to each PDB specific needs using the DBMS_CONNECTION_POOL subprograms. Processes are not shared between PDBs. ROOT and PDBs only share metadata. They do not share data in pool configuration tables. Each container stores its own pool configuration data. The statistics in the views will be kept for each PDB. The PDB administrator can neither alter the broker parameters, such as MINSIZE, MAXCONN_CBROK and NUM_CBROK, nor can they set the broker parameters to 2147483647. These parameters can only be modified in the database parameter CONNECTION_BROKERS and apply to the entire CDB. The database parameters MIN_AUTH_SERVERS, MAX_AUTH_SERVERS and DRCP_DEDICATED_OPT are dynamically per PDB modifiable using the ALTER SYSTEM command.

To avoid configuration, by default, DRCP auto-starts when a PDB opens, and it stops when the PDB closes. If the PDB administrator stops the pool, then the DRCP does not auto-start when the PDB re-opens in the future.

The DRCP_DEDICATED_OPT parameter controls the use of dedicated optimization with DRCP. It is disabled by default when per-PDB mode is enabled.

The MAX_TXN_THINK_TIME parameter specifies the maximum time of inactivity allowed before termination for a client with an open transaction. This can be set to allow more time than the client that does not have transactions (specified by the MAX_THINK_TIME parameter value). This allows efficient pool reuse, while giving incomplete transactions a longer time to conclude.

See Also

- *Oracle Database PL/SQL Packages and Types Reference* for details about the DBMS_CONNECTION_POOL package
- *Oracle Database Administrator's Guide* for more information about Configuring the Connection Pool for Database Resident Connection Pooling
- *Oracle Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Universal Connection Pool for JDBC Developer's Guide*

3.10.2.1 Managing Permissions for Per-PDB DRCP

For the per-PDB DRCP configuration (ENABLE_PER_PDB_DRCP=TRUE), the SYS user must grant permissions to the PDB administrator to manage DRCP at the PDB level.

By default, DRCP is configured at the CDB level. In the CDB DRCP mode, a single DRCP pool running in the CDB is shared across all the PDBs. In the CDB DRCP mode, the

Copyright © 1999, 2025, Oracle and/or its affiliates.

ENABLE_PER_PDB_DRCP database initialization parameter is set to FALSE. The ENABLE_PER_PDB_DRCP parameter can be set to TRUE to enable per-PDB DRCP.

For a PDB administrator to access the DBMS_CONNECTION_POOL package and query the DRCP statistics, the ROOT user (SYS) must grant the following permissions to the PDB administrator (PDB1ADMIN in this case).

```
GRANT CREATE SESSION, CREATE SYNONYM TO PDB1ADMIN;
GRANT EXECUTE ON DBMS_CONNECTION_POOL TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_STATS TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_CC_STATS TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_CONN_INFO TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_CC_INFO TO PDB1ADMIN;
GRANT SELECT ON V_$AUTHPOOL_STATS TO PDB1ADMIN;
```

To make the management and monitoring of DRCP pools easier, the PDB administrator (PDB1ADMIN in this case) can create the following synonyms.

```
CREATE SYNONYM DBMS_CONNECTION_POOL FOR SYS.DBMS_CONNECTION_POOL;
CREATE SYNONYM V$CPOOL_STATS FOR SYS.V_$CPOOL_STATS;
CREATE SYNONYM V$CPOOL_CC_STATS FOR SYS.V_$CPOOL_CC_STATS;
CREATE SYNONYM V$CPOOL_CONN_INFO FOR SYS.V_$CPOOL_CONN_INFO;
CREATE SYNONYM V$CPOOL_CC_INFO FOR SYS.V_$CPOOL_CC_INFO;
CREATE SYNONYM V$AUTHPOOL_STATS FOR SYS.V_$AUTHPOOL_STATS;
```

Once this is done, pool management is allowed only at the PDB level by the respective PDB administrators.

3.10.3 Using Multi-pool DRCP

Starting Oracle AI Database 26ai, you can use multiple, named DRCP pools. Database administrators can add, configure, manage, monitor, or remove a DRCP pool at the PDB or CDB level. You can configure DRCP to use connections (pooled servers) from any available DRCP pool and have a specific application acquire connections from a configured DRCP pool.

The default system-named pool: SYS_DEFAULT_CONNECTION_POOL is always available. You can create a new, named pool using the ADD_POOL procedure in the DBMS_CONNECTION_POOL package. Depending on your requirements, applications can use connections from any DRCP pool.

Having multiple pools allows finer control over the DRCP pool usage. You can have pooled servers available to a few applications or services at all times. You can avoid a situation where connections from some applications occupy all the pooled servers of a DRCP pool while other applications wait for an available pooled server in that pool.

See Also

- Adding a DRCP Pool
- Removing a DRCP Pool
- Connecting to a Pool in Multi-pool DRCP

Copyright © 1999, 2025, Oracle and/or its affiliates.

Components of Multi-pool DRCP

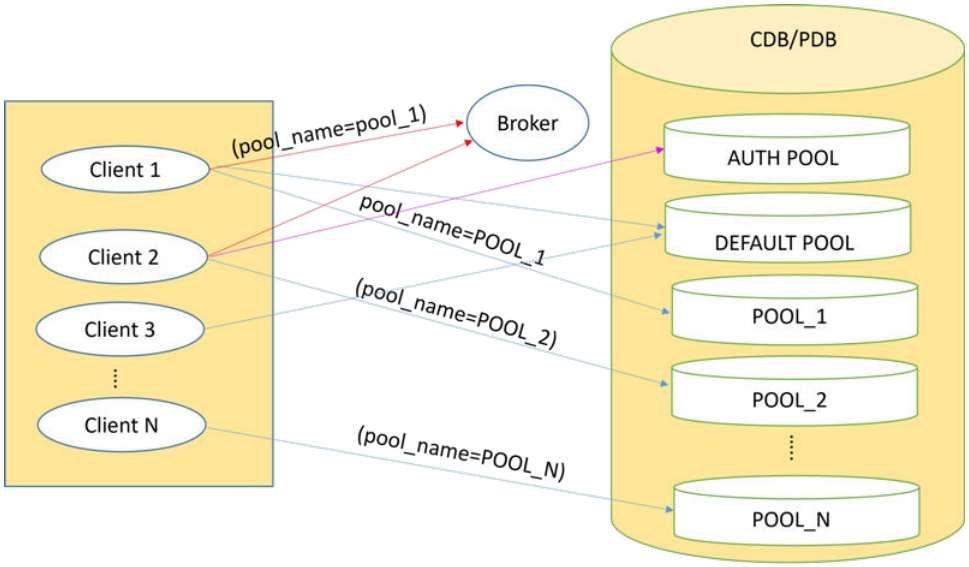
The following components are common to all the multi-pool DRCP pools and the default pool, at the PDB or CDB level.

- A connection broker to manage the pooled servers and handle the connection hand-off process.
- An authentication pool to authenticate user connections when client applications connect to DRCP.

Other components include:

- A default pool called `SYS_DEFAULT_CONNECTION_POOL` to handle DRCP for pools when no pool name is specified. The client cannot add or remove the default pool.
- Multiple DRCP pools that are added at the PDB or CDB level.

Figure 3-1 Multi-pool DRCP



3.10.3.1 Adding a DRCP Pool

By connecting to a PDB, a PDB administrator can add a DRCP pool at the PDB level. Similarly, a CDB administrator can add a DRCP pool at the CDB level.

To add a new pool, use the `dbms_connections_pool.add_pool()` procedure.

> **See Also**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package

You must add a unique name for the pool. For example, you cannot add a new pool called `SYS_DEFAULT_CONNECTION_POOL` because it is the default pool name.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Multi-pool DRCP has the same default configuration values as the per-PDB DRCP, such as `minsize=0`, `num_cbrok=0`, and `maxconn_cbrok=0`. If the configuration is known at the time of adding the pool, you can use the `dbms_connections_pool.add_pool()` procedure to set the configuration while adding the pool itself. If the configuration is not known, then it must be reconfigured later using the `configure_pool()` procedure with the new pool name and its configuration values.

You can use the `start_pool()`, `stop_pool()`, `configure_pool()`, `alter_param()` and `restore_defaults()` procedures of the `dbms_connection_pool` package for the newly added pools.

The `V$CPOOL_STATS`, `V$CPOOL_CC_STATS`, `V$CPOOL_CONN_INFO`, and `V$CPOOL_CC_INFO` views have the `POOL_NAME` columns and provide the statistics on the added pools.

Example 3-1 Adding a DRCP pool

```
exec dbms_connection_pool.add_pool('mypool')
```

3.10.3.2 Removing a DRCP Pool

You must be a PDB or CDB administrator to remove a DRCP pool.

To remove a DRCP pool, use the `dbms_connections_pool.remove_pool()` procedure.

> **See Also**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package

You cannot use the `remove_pool` procedure to remove:

- The default pool.
- A pool that is currently active. You must stop the pool before removing it.
- A pool that does not exist.

Example 3-2 Removing a DRCP pool

```
exec dbms_connection_pool.remove_pool('mypool')
```

3.10.3.3 About Authentication Pool in Multi-pool DRCP

An authentication pool is common to all the added pools in multi-pool DRCP and the default pool (`SYS_DEFAULT_CONNECTION_POOL`) for CDB-wide and per-PDB pools. When the first pool in a multi-pool DRCP starts functioning, the authentication pool also starts functioning. The authentication pool stops functioning when the last `ACTIVE` pool stops functioning.

The statistics on `num_authentications` (number of authentications) for each pool is maintained in the `v$cpool_stats` and `v$cpool_cc_stats` views. The statistics about the authentication pool is provided in the `V$AUTHPOOL_STATS` view.

3.10.3.4 Managing the Connection Broker in Multi-pool DRCP

The ROOT container owns the connection broker, hence, with per-PDB DRCP, a broker process is always running. For CDB-wide pools, the database administrator can configure brokers using the `DBMS_CONNECTION_POOL` package but because the broker is at the ROOT-

Copyright © 1999, 2025, Oracle and/or its affiliates.

level and common to all the pools, changing the number of brokers for one pool affects all the other pools, which is fine as DBA is the same for all pools, unlike per-PDB pools. With per-PDB pools, a PDB administrator cannot alter or configure brokers using the `DBMS_CONNECTION_POOL` package because it affects other per-PDB pools. Therefore, it is recommended that for per-PDB pools, a CDB administrator must set brokers using the `CONNECTION_BROKERS` in the database initialization parameter.

Example:

```
alter system set CONNECTION_BROKERS='((TYPE=POOLED)(BROKERS=1)(CONNECTIONS=40000))';
```

3.10.4 Sharing Proxy Sessions

Starting with Oracle Database 12c Release 2 (12.2.0.1), proxy sessions in DRCP are shared among applications that are connected to the same schema.

3.10.5 Using JDBC with DRCP

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.

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
```

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- Starting Database Resident Connection Pool
- *Oracle Database JDBC Developer's Guide* for more information about APIs that are used to control custom connection pool implementations
- *Oracle Database JDBC Developer's Guide* for more information about enabling DRCP on client side

3.10.6 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.

To release a session to the OCI session pool for DRCP, an OCI application invokes `OCISessionRelease()`.

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 `authInfop` parameter set to `OCI_ATTR_CONNECTION_CLASS` and specifying a connection class name or by using the `OCIAuthInfo` handle before invoking `OCISessionGet()`.

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

See Also

- *Oracle Call Interface Programmer's Guide* for information about `OCISessionGet()`
- *Oracle Call Interface Programmer's Guide* for more information about `OCISessionRelease()`

3.10.7 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 3-3 Setting Session Purity for New Session

This example shows how a connection pooling application sets up a new session.

```
/* OCIAttrSet method */
ub4 purity = OCI_ATTR_PURITY_NEW;
OCIAttrSet(authInfop, OCI_HTYPE_AUTHINFO, &purity, (ub4)sizeof(purity) OCI_ATTR_PURITY,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
errhp);

/*  OCISessionGet mode method */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0, NULL,
NULL, NULL, OCI_SESSGET_SPOOL);
/* 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.

3.10.8 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, you can 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 (*).

3.10.8.1 Example: Setting the Connection Class as HRMS

You can use the `OCISessionPoolCreate` API to set a connection class as HRMS.

Example 3-4 specifies that an HRMS application needs sessions with the connection class `HRMS`.

Example 3-4 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);

OCIAttrSet (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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

3.10.9 Session Purity and Connection Class Defaults

Table 3-5 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 3-5 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
`OCI_ATTR_PURITY`	`OCI_ATTR_PURITY_SELF`	`OCI_ATTR_PURITY_NEW`
`OCI_ATTR_CONNECTION_CLASS`	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

3.10.10 Setting the Purity and Connection Class in the Connection String

You can specify the connection class and purity attributes in the `CONNECT_DATA` section of the Connect strings.

When it is not possible to change the application setting in the application code, you can override the value set in the application by setting the parameters: `POOL_CONNECTION_CLASS` and `POOL_PURITY` in the Connect string.

> **Note**
>
> If the `POOL_PURITY` is specified as `SELF` in the Connect string, applications that explicitly get `NEW` purity connections from the OCI Session Pool do not drop the DRCP pooled session while releasing the connection back to the OCI Session Pool, even if they specify the `OCI_SESSRLS_DROPSESS` mode. Such applications should continue to use the programmatic way of specifying the purity.

> **See Also**
>
> - *Oracle Database Net Services Reference* for more information about DRCP parameters

3.10.11 Starting DRCP

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Database Administrator's Guide* for detailed information about configuring the pool

3.10.12 Shut Down Connection Draining for DRCP

You can shut down (close) a DRCP pool using the `DBMS_CONNECTION_POOL.STOP_POOL()` procedure in either of the following ways.

- If a DRCP pool is idle and inactive after your application has released all the connections back to the DRCP pool, use the `STOP_POOL` procedure using the pool name as the parameter to close the idle or inactive pool.
- If a DRCP pool has active connections, use an optional `draintime` parameter in the `STOP_POOL` procedure to close the connections and DRCP pool without waiting for the DRCP pool to go idle and inactive.

You can use the `draintime` parameter in the `STOP_POOL` procedure to close an active DRCP pool immediately, or after a specified connection drain time. The `draintime` parameter indicates how many seconds a DRCP pool is allowed to remain active before the pool is drained of its connections and the pool is closed.

For example, to allow 20 seconds for active connections in a DRCP pool to complete their tasks before being closed, run the following:

```
DBMS_CONNECTION_POOL.STOP_POOL('my_pool', 20);
```

To close a DRCP pool immediately, set the `draintime` parameter to 0, as follows:

```
DBMS_CONNECTION_POOL.STOP_POOL('my_pool', 0);
```

The ability to close active connection pools provides the DBAs better control over the DRCP usage and configurations.

> **See Also**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package

3.10.13 Enabling DRCP

To enable DRCP in an application, specify either `:POOLED` in the Easy Connect string (as in Example 3-5) or `(SERVER=POOLED)` in the TNS Connect string (as in Example 3-6).

Example 3-5 Enabling DRCP With `:POOLED` in the Easy Connect string

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 3-6 Enabling DRCP With SERVER=POOLED in the TNS Connect string

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

3.10.14 Connecting to a Pool in Multi-pool DRCP

You can specify a pool name for each connection.

To access multi-pool DRCP and to connect with an appropriate pool, specify POOL_NAME=<pool_name> along with SERVER=POOLED in the connect string:

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=<host>)(PORT=<port>))
        (CONNECT_DATA=(SERVER=POOLED)(POOL_NAME=<pool_name>)))
```

> **Note**
>
> The connect string must include SERVER=POOLED for POOL_NAME=<pool_name> to work.

An application can connect to any of the available pools. DRCP checks if the pool name (<pool_name>) in the connect string exists, and if it does, the connection uses the pooled server with the given pool name. If the pool name does not exist, an error is returned. If no pool name is specified in the connect string, the connection is handed off to the default pool, provided it is active.

> **See Also**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_CONNECTION_POOL package
> - Using Multi-pool DRCP

Example 3-7 Connecting to a Pool in Multi-pool DRCP

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=phoenix92128)(PORT=1521))
        (CONNECT_DATA=(SERVICE_NAME=cdb1_pdb1.regress.rdbms.dev.us.oracle.com)
        (SERVER=POOLED)(POOL_NAME=mypool)))
```

3.10.15 Implicit Connection Pooling

Implicit connection pooling is aimed at enhancing the benefits derived from DRCP and Proxy Resident Connection Pooling (PRCP). With implicit connection pooling, a database session from DRCP/PRCP gets automatically mapped to and unmapped from an application connection at runtime. For example, for a transaction, an application implicitly uses or reuses an available connection from the connection pool and releases the connection back to the pool after the transaction is complete.

> **See Also**
>
> - *Implicit Connection Pooling with CMAN-TDM and PRCP* below for more information about PRCP.

Copyright © 1999, 2025, Oracle and/or its affiliates.

You can use implicit connection pooling without making any application-level change or calling the pooling APIs. The configuration for implicit connection pooling is performed on the client side. You must provide the `POOL_BOUNDARY` parameter in the `CONNECT_DATA` section of the connect string. After configuring the `POOL_BOUNDARY` parameter on DRCP/PRCP, session mapping or unmapping is performed based on the session state. Implicit connection pooling uses a session's state and specific application settings to automatically detect the time (boundary) to unmap and release a connection back to the pool.

See Also

- *Implicit Stateful and Stateless Sessions* below for more information about session states.

Implicit connection pooling provides better scalability and enables efficient use of database resources for applications that do not use application connection pools, such as Oracle Call Interface (OCI) Session Pool or Java Database Connectivity (JDBC) Oracle Universal Connection Pool (UCP).

Implicit connection pooling is beneficial to on-premise and cloud applications in the following cases:

- Applications that directly connect to Oracle Database and have pooling requirements but do not leverage the server-side connection pooling APIs of Oracle.
- Applications that are connected to Connection Manager-Traffic Connector Mode (CMAN-TDM) in PRCP and have pooling requirements but do not use the connection pooling APIs. These applications can configure TDM in PRCP mode to use Implicit Connection Pooling.
- Applications that use connection pooling APIs but need to further optimize the use of shared resources. Implementing implicit connection pools provides better scalability because applications do not hold up sessions unnecessarily while waiting for an explicit API call.
- Middle-tier servers can use Implicit Connection Pooling for better scalability through implicit multiplexing of database sessions across a larger number of middle-tier connections.

Implicit connection pooling is available to clients on Oracle AI Database 26ai, and in applications that use any data access drivers, such as OCI, JDBC, ODP.Net, cx_Oracle (Python), node-oracledb (Node.js), PHP-OCI8, Pre-compilers, ODBC, and OCCI.

Topics

- Implicit Stateful and Stateless Sessions
- Statement and Transaction Boundary
- Configuring Implicit Connection Pool Boundaries
- Impact of Round-trip OCI Calls on Implicit Connection Pooling States
- Deciding which Pool Boundary to Use
- Implicit Connection Pooling with CMAN-TDM and PRCP
- Setting or Resetting the Session State at the Boundaries During Deployment
- Using the Session Cached Cursors with Implicit Connection Pooling
- Security

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.10.15.1 Implicit Stateful and Stateless Sessions

A session is implicitly stateless if an active session's state satisfies all the following conditions:

- All the cursors that are open in the session have been fetched through to completion.
- The session has no active transactions.
- The session has no temporary LOBs.
- The session has no global temporary tables with rows.
- The session does not have private temporary tables open.

If a session state does not satisfy any of the aforesaid conditions, the session is implicitly stateful.

3.10.15.2 Statement and Transaction Boundary

Implicit Connection Pooling uses time boundaries to release a session back to the connection pool. A time boundary is a point in time in the life cycle of the application when an application session is released back to the pool. The pool could either be a DRCP pool or PRCP pool.

The two boundaries used in Implicit Connection Pooling are Statement Boundary and Transaction Boundary.

- Statement Boundary is used to release a session back to the connection pool when the session is implicitly stateless.

  A session is implicitly stateless when all open cursors in a session have been fetched through to completion, and there are no active transactions, temporary tables, or temporary LOBs.
- Transaction Boundary is used to release a session back to the connection pool when a transaction ends implicitly or explicitly, or when a transaction is not available and the session is stateless.

> **Note**
>
> – The release to the connection pool closes any active cursors, temporary tables, and temporary LOBs.
>
> – After a statement is executed or a temporary LOB is created and a commit or rollback is performed within the transaction boundary:
>
> * Subsequent fetch operations may encounter an invalid cursor-related error such as ORA-01001.
> * Subsequent use of temporary LOB may encounter an error such as ORA-22922.
>
> Working with a persistent LOB can continue even after an implicit release. However, if an implicit operation lands on a different instance, an ORA-43887 error may occur. In such cases, the operation should be re-attempted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.10.15.3 Configuring Implicit Connection Pool Boundaries

In the application tier, to enable DRCP pooling, the client must specify the server type as POOLED (SERVER=POOLED) in the connection string of the `tnsnames.ora` file.
To configure the Implicit Connection Pooling boundary for DRCP on the server side or for PRCP in the TDM mode, add the POOL_BOUNDARY attribute in the connection string. The attribute takes two valid values, namely STATEMENT and TRANSACTON, and has no default value. If the POOL_BOUNDARY attribute is not included in the connection string, Implicit Connection Pooling is disabled.

> **Note**
>
> If the application provides the POOL_BOUNDARY=STATEMENT or POOL_BOUNDARY=TRANSACTION attribute in the connection string without providing the SERVER=POOLED attribute, Implicit Connection Pooling is disabled and the POOL_BOUNDARY directive is ignored.

> **Tip**
>
> Use the Net Configuration Assistant (netca) utility to add the POOL_BOUNDARY attribute into the connection string.

- To specify the Statement Boundary, use POOL_BOUNDARY=STATEMENT in the Easy Connect string or in the TNS Connect string, as shown in the following examples:

  > **Note**
  >
  > The Easy Connect string syntax is supported for Oracle AI Database 26ai, and later releases.

```
inst1=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=STATEMENT)))

host:port/servicename:pooled?pool_boundary=statement
```

- To specify the Transaction Boundary, use POOL_BOUNDARY=TRANSACTIION in the Easy Connect string or in the TNS Connect string, as shown in the following examples:

```
inst1=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=TRANSACTION)))

host:port/servicename:pooled?pool_boundary=transaction
```

3.10.15.4 Impact of Round-trip OCI Calls on Implicit Connection Pooling States

If an application uses the session pooling APIs along with the POOL_BOUNDARY=STATEMENT or TRANSACTION attribute in the connection string, then the connection string settings take

Copyright © 1999, 2025, Oracle and/or its affiliates.

precedence over the pooling APIs. The session is released back to the DRCP or PRCP pool using the statement or transaction boundary directive, overriding the session release API call. For instance, if an `OCISessionRelease()` call is made on a session that was implicitly released using the `POOL_BOUNDARY=STATEMENT` or `TRANSACTION` attribute in the connection string, then the `OCISessionRelease()` call is a no-op. Conversely, if an `OCISessionRelease()` call is made when the session is implicitly stateful, and hence, not implicitly released, then the session is released back to the pool based on the API call.

> **Note**
>
> For all the `POOL_BOUNDARY` options, the default purity is set to `SELF`. You can specify the purity using the `POOL_PURITY` parameter in the connect string to override the default purity value.

> **See Also**
>
> *Oracle Call Interface Programmer's Guide* for more information about pooling options and OCI round-trip calls

3.10.15.5 Deciding which Pool Boundary to Use

Use `POOL_BOUNDARY` with `TRANSACTION` for applications that create many session states from partially fetched cursors, temporary LOBs, and global or private temporary tables and for applications that have occasional commits or rollbacks of transactions.

Use `POOL_BOUNDARY` with `STATEMENT` for applications that create minimal session states from partially fetched cursors, temporary LOBs, and global or private temporary tables.

3.10.15.6 Implicit Connection Pooling with CMAN-TDM and PRCP

Implicit Connection Pooling enables applications to leverage PRCP if Oracle Connection Manager (CMAN) is set to Traffic Director Mode (TDM). For applications that connect through the CMAN-TDM connection proxy, Implicit Connection Pooling with PRCP can help maximize the pooled server usage and reduce the server resource usage.

CMAN is a connection concentrator from Oracle Net Services that enables applications to connect to Oracle Database over the Cloud. TDM is an added layer on top of CMAN that provides a shield to the applications from database instance outages and takes care of failed connections due to outages.

You can configure CMAN to operate in the TDM mode. CMAN-TDM is an Oracle Database connection proxy that enables a client application to connect to an Oracle Database (on-premises and cloud) without exposing the underlying database details to the client.

In the default mode of operation, CMAN-TDM creates one connection to the database for each incoming connection from the client. Idling client sessions can unnecessarily keep database connections engaged. To avoid the session idling, you can configure CMAN-TDM in the PRCP mode. In the PRCP mode, CMAN-TDM maintains an OCI session pool (OCISessionPool) and behaves like DRCP. When an application thread requires to interact with the database on a connection, CMAN-TDM picks up a session from the OCISessionPool and maps the incoming connection to the session. When the application thread indicates that it is done with the database activity, the connection is handed back (or unmapped) to the CMAN gateway process. The outgoing session is then released back to the OCISessionPool in CMAN-TDM.

Copyright © 1999, 2025, Oracle and/or its affiliates.

As a result, CMAN-TDM has fewer processes and sessions on the database than the connections.

Use the following connect descriptor in `tnsnames.ora` file to switch different connect modes for Implicit Connection Pooling. The connection string can point either to DRCP or to CMAN-TDM in the PRCP mode:

```
inst1s=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=STATEMENT)))

inst1t=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=TRANSACTION)))
```

In the above descriptors, the `CONNECT_DATA` section contains the following new attribute:

- For `inst1s`, the `POOL_BOUNDARY=STATEMENT` parameter specifies that implicit pooling semantics apply on a statement boundary basis.
- For `inst1t`, the `POOL_BOUNDARY=TRANSACTION` parameter specifies that implicit pooling semantics apply on a transaction boundary basis.

3.10.15.7 Setting or Resetting the Session State at the Boundaries During Deployment

If you have application administrator privileges with invoker rights, you must define a package called `ORA_CPOOL_STATE` once for every application user. Within the package, you can define a procedure called `ORA_CPOOL_STATE_CALLBACK` having two VARCHAR2 parameters, namely `connection_class` and `service`. After the package is defined, any implicit get call on the server invokes the package and procedures defined in it.
Implicit Connection Pooling works seamlessly with connection class. If applications use connection class, then every implicit get call that happens on the server, CMAN-TDM, or both, honors the connection class setting.

> **Note**
>
> If your application alters the session parameters that affect a query or DML (for example, while dealing with timestamp values) and you want to set the state as needed, you must move the `ALTER SESSION` logic to the PL/SQL callbacks. This is required because if your application runs an `ALTER SESSION` statement followed by the query execution, it might run on two different pooled servers with implicit pooling.
>
> Alternatively, you can have the application run the `ALTER SESSION` statement and the queries together in an anonymous PL/SQL block.

Here is an example that uses only the `connection_class` and `service` parameters in its fixup logic.

```
CREATE OR REPLACE PACKAGE ORA_CPOOL_STATE authid current_user AS
PROCEDURE ORA_CPOOL_STATE_GET_CALLBACK(varchar2 in service, varchar2 in
connection_class);
PROCEDURE ORA_CPOOL_STATE_RLS_CALLBACK(varchar2 in service, varchar2 in
connection_class);
END;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
/
CREATE OR REPLACE PACKAGE BODY ORA_CPOOL_STATE AS
PROCEDURE ORA_CPOOL_STATE_GET_CALLBACK(varchar2 in service, varchar2 in
connection_class)
IS
BEGIN
  IF (connection_class = 'GERMAN') THEN
    ALTER SESSION SET NLS_CURRENCY='€';
  ELSE IF (connection_class = 'INDIAN') THEN
    ALTER SESSION SET NLS_CURRENCY='';
  END IF;
END;

PROCEDURE ORA_CPOOL_STATE_RLS_CALLBACK(varchar2 in service, varchar2 in
connection_class)
IS
BEGIN
/* clear all the session state by restoring to defaults */
  IF (service = 'HR') THEN
    ALTER SESSION SET NLS_CURRENCY='$';
    ALTER SESSION SET date_format='';
END;
END;
```

3.10.15.8 Using the Session Cached Cursors with Implicit Connection Pooling

Implicit Connection Pooling clears the statement cache each time a session is implicitly released. As a result, multiple executions of the same query work as fully executed, separate queries instead of re-executed, same queries. Use session cached cursors to compensate for these shortcomings. Add the following to set session cached cursors in the `init.ora` parameter file:

```
session_cached_cursors=20
```

3.10.15.9 Security

For security, Implicit Connection Pooling ensures that a user session implicitly released to the DRCP or PRCP pool is available for requests only for the same user. If a current connected user has defined the `ORA_CPOOL_STATE` package and the `ORA_CPOOL_STATE_CALLBACK` callback, the package and the callback execute only in the context of the current connected user. A connected user, say Scott, cannot execute a callback of another user, say HR.

Related Topics

- *Oracle Call Interface Developer's Guide*
- *Oracle C++ Call Interface Developer's Guide*
- *Oracle AI Database JDBC Developer’s Guide*
- *Oracle Universal Connection Pool Developer’s Guide*
- *Oracle Data Provider for .NET Developer's Guide*

3.10.16 Benefiting from the Scalability of DRCP in an OCI Application

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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

`OCISessionPoolCreate()`

`OCISessionGet()`

`OCISessionRelease(),`

`OCISessionPoolDestroy()`

3.10.17 Benefiting from the Scalability of DRCP in a Java Application

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).

See Also

- Benefiting from the Scalability of DRCP in an OCI Application for more information about how Java applications benefit from DRCP as OCI applications

3.10.18 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 3-8 shows a DRCP application.

See Also

- *Oracle Call Interface Programmer's Guide*

Example 3-8 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 3-9 and Example 3-10 show connect strings that deploy code in 10 middle-tier hosts that service the `BOOKSTORE` application from Example 3-8.

In Example 3-9, assume that the database is Oracle Database 12*c* (or earlier) in dedicated server mode with DRCP not enabled and that the client has 12*c* libraries. The application gets dedicated server connections from the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 3-9 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 3-10, assume that DRCP is enabled on the Oracle Database 12*c* 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 3-10 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)))
```

3.10.19 Compatibility and Migration

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

- An Oracle Database 12*c* database with DRCP disabled
- A database server from a release earlier than Oracle Database 12*c*
- An Oracle Database 12*c* 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.

See Also

- *Oracle Database JDBC Developer's Guide* for more information about Oracle JDBC drivers support for DRCP

3.10.20 Using DRCP with Oracle Database Native Network Encryption

You can use the following supported checksum algorithms and encryption algorithms while using DRCP with Oracle Database Native Network Encryption.

Checksum Algorithms

- SHA1
- SHA256
- SHA384
- SHA512

Encryption Algorithms

- AES128
- AES192
- AES256

The DES, DES40, 3DES112, 3DES168, MD5, RC4_40, RC4_56, RC4_128, and RC4_256 algorithms are deprecated in this release. To transition your Oracle Database environment to

Copyright © 1999, 2025, Oracle and/or its affiliates.

use stronger algorithms, download and install the patch described in My Oracle Support note 2118136.2.

3.10.21 DRCP Restrictions

The following cannot be performed when connected to a DRCP pooled server:

- 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 TCPS or IPC connections
- Using enterprise user security with DRCP
- Using migratable sessions on the server side, either directly (using the `OCI_MIGRATE` option) or indirectly (invoking `OCIConnectionPoolCreate()`)
- Creating multiple sessions on a DRCP server for session switching or for dual session proxy
- Using initial client roles
- Using application context attributes (such as `OCI_ATTR_APPCTX_NAME` and `OCI_ATTR_APPCTX_VALUE`)

Attempting any of the aforementioned may result in the following error: ORA-56609: Usage not supported 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.

You can use Application Continuity with DRCP but you must ensure that the sessions are returned to the pool with the default session state.

> **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 AES128 for another set of connections, then the application must specify different connection classes for each set.

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.10.22 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. 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.

> **Note**
>
> An application that specifies the attribute `OCI_ATTR_SESSION_STATE` or `OCI_SESSION_STATELESS` must also specify session purity and connection class.

> **See Also**
>
> - *Oracle Call Interface Programmer's Guide* for more information about `OCI_ATTR_SESSION_STATE` attribute

3.10.23 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**
>
> Using DRCP with Custom Pools

Example 3-11 shows a code fragment that explicitly marks session states.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 3-11 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`

3.10.24 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.

3.10.25 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

  - 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.

3.11 Memoptimize Pool

This pool optimizes the read operation for select statements

Memoptimize Rowstore performs high-performance reads for tables specified with the `MEMOPTIMIZE FOR READ` clause.

> **Note**
>
> Deferred inserts cannot be rolled back because they do not use standard locking and redo mechanisms

Related Topics

- *Oracle AI Database Concepts*
- *Oracle AI Database PL/SQL Packages and Types Reference*
- *Oracle AI Database SQL Language Reference*
- *Oracle AI Database Reference*

3.12 Oracle RAC Sharding

This section explains about Oracle RAC Sharding

Oracle RAC Sharding increases the performance and scalability of a Oracle RAC database with minimal application changes. It affinitizes table partitions to Oracle RAC instances, and routes the database requests, which specify a partitioning key to the instance that logically holds the corresponding partition. This provides better cache utilization and reduces block pings across instances. The partitioning key can be added only to the most performance critical requests. Requests that do not specify the key works transparently and can be routed to any instance. This feature can be enabled by executing an `ALTER SYSTEM` command and without modifying the database schema and SQL statements.

> **Note**
>
> The partitioning key value must be provided when requesting a database connection.

Related Topics

- *Oracle AI Database Net Services Administrator's Guide*
- *Oracle Call Interface Developer's Guide*
- *Oracle Universal Connection Pool Developer’s Guide*
- *Oracle Data Provider for .NET Developer's Guide*
- *Oracle Real Application Clusters Administration and Deployment Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

- *Oracle AI Database Concepts*
- *Oracle AI Database Administrator's Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

4

Designing Applications for Oracle Real-World Performance

When you design applications for real world performance, you should consider how code for bind variables, instrumentation, and set-based processing.

Topics:

- Using Bind Variables
- Using Instrumentation
- Using Set-Based Processing

4.1 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.

A 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—

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

4.2 Using Instrumentation

To use instrumentation means adding 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

SQL Trace Facility (SQL_TRACE) for more information

4.3 Using Set-Based Processing

A common task in database applications in a data warehouse environment is querying or modifying a huge data set.

For example, an application might join data sets numbering in the tens of millions of rows, filter on a set of criteria, perform aggregations, and then display the result to the user. Alternatively, an application might filter out rows from one billion-row table based on specified criteria, and then insert matching rows into another table.

The problem for application developers is how to achieve high performance when processing these large data sets. Processing techniques fall into two categories: iterative, and set-based. Over years of testing, the Oracle Real-World Performance group has discovered that set-based processing techniques perform orders of magnitude better for database applications that process large data sets.

Topics:

- Iterative Data Processing
- Using Set-Based Processing

4.3.1 Iterative Data Processing

Iterative data processing processes data row by row, using arrays, or using manual parallelism.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Topics:

- About Iterative Data Processing
- Iterative Data Processing: Row-By-Row
- Iterative Data Processing: Arrays
- Iterative Data Processing: Manual Parallelism

4.3.1.1 About Iterative Data Processing

In this type of processing, applications use conditional logic to iterate through a set of rows.

You can write iterative applications in PL/SQL, Java, or any other procedural or object-oriented language. The technique is "iterative" because it breaks the row source into subgroups containing one or more rows, and then processes each subgroup. A single process can iterate through all subgroups, or multiple processes can iterate through the subgroups in parallel.

Typically, although not necessarily, iterative processing uses a client/server model as follows:

1. Transfer a group of rows from the database server to the client application.
2. Process the group within the client application.
3. Transfer the processed group back to the database server.

You can implement iterative algorithms using three main techniques: row-by-row processing, array processing, and manual parallelism. Each technique obtains the same result, but from a performance perspective, each has its benefits and drawbacks.

4.3.1.2 Iterative Data Processing: Row-By-Row

Of the iterative techniques, row-by-row processing is the most common.

A single process loops through a data set and operates on a single row a time. In a typical implementation, the application retrieves each row from the database, processes it in the middle tier, and then sends the row back to the database, which executes DML and commits.

Assume that your functional requirement is to query an external table named `ext_scan_events`, and then insert its rows into a heap-organized staging table named `stage1_scan_events`. The following PL/SQL block uses a row-by-row technique to meet this requirement:

```
declare
  cursor c is select s.* from ext_scan_events s;
  r c%rowtype;
begin
  open c;
  loop
    fetch c into r;
    exit when c%notfound;
    insert into stage1_scan_events d values r;
    commit;
  end loop;
  close c;
end;
```

The row-by-row code uses a cursor loop to perform the following actions:

1. Fetch a single row from `ext_scan_events` to the application running in the client host, or exit the program if no more rows exist.

Copyright © 1999, 2025, Oracle and/or its affiliates.

2. Insert the row into `stage1_scan_events`.
3. Commit the preceding insert.
4. Return to Step 1.

The row-by-row technique has the following advantages:

- It performs well on small data sets. Assume that `ext_scan_events` contains 10,000 records. If the application processes each row in 1 millisecond, then the total processing time is 10 seconds.
- The looping algorithm is familiar to all professional developers, easy to write quickly, and easy to understand.

The row-by-row technique has the following disadvantages:

- Processing time can be unacceptably long for large data sets. If `ext_scan_events` contains 1 billion rows, and if the application processes each row in an average of 1 miliseconds, then the total processing time is 12 days. Processing a trillion-row table requires *32 years*.
- The application executes serially, and thus cannot exploit the native parallel processing features of Oracle Database running on modern hardware. For example, the row-by-row technique cannot benefit from a multi-core computer, Oracle RAC, or Oracle Exadata Machine. For example, if the database host contains 16 CPUs and 32 cores, then 31 cores will be idle when the sole database server process reads or write each row. If multiple instances exist in an Oracle RAC deployment, then only one instance can process the data.

4.3.1.3 Iterative Data Processing: Arrays

Array processing is identical to row-by-row processing, except that it processes a group of rows in each iteration rather than a single row.

Like the row-by-row technique, array processing is serial, which means that only one database server process operates on a group of rows at one time. In a typical array implementation, the application retrieves each group of rows from the database, processes it in the middle tier, and then sends the group back to the database, which performs DML for the group of rows, and then commits.

Assume that your functional requirement is the same as in the example in Iterative Data Processing: Row-By-Row: query an external table named `ext_scan_events`, and then insert its rows into a heap-organized staging table named `stage1_scan_events`. The following PL/SQL block, which you execute in SQL*Plus on a separate host from the database server, uses an array technique to meet this requirement:

```
declare
  cursor c is select s.* from ext_scan_events s;
  type t is table of c%rowtype index by binary_integer;
  a t;
  rows binary_integer := 0;
begin
  open c;
  loop
    fetch c bulk collect into a limit array_size;
    exit when a.count = 0;
    forall i in 1..a.count
      insert into stage1_scan_events d values a(i);
    commit;
  end loop;
  close c;
end;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The preceding code differs from the equivalent row-by-row code in using a `BULK COLLECT` operator in the `FETCH` statement, which is limited by the array_size value of type `PLS_INTEGER`. For example, if array_size is set to 100, then the application fetches rows in groups of 100.

The cursor loop performs the following sequence of actions:

1. Fetch an array of rows from ext_scan_events to the application running in the client host, or exit the program when the loop counter equals 0.
2. Loop through the array of rows, and insert each row into the `stage1_scan_events` table.
3. Commit the preceding inserts.
4. Return to Step 1.

In PL/SQL, the array code differs from the row-by-row code in using a counter rather than the cursor attribute `c%notfound` to test the exit condition. The reason is that if the fetch collects the last group of rows in the table, then `c%notfound` forces the loop to exit, which is undesired behavior. When using a counter, each fetch collects the specified number of rows, and when the collection is empty, the program exits.

The array technique has the following advantages over the row-by-row technique:

- The array enables the application to process a group of rows at the same time, which means that it reduces network round trips, `COMMIT` time, and the code path in the client and server. When combined, these factors can potentially reduce the total processing time by an order of magnitude
- The database is more efficient because the server process batches the inserts, and commits after every group of inserts rather than after every insert. Reducing the number of commits reduces the I/O load and lessens the probability of log sync wait events.

The disadvantages of this technique are the same as for row-by-row processing. Processing time can be unacceptable for large data sets. For a trillion-row table, reducing processing time from 32 years to 3.2 years is still unacceptable. Also, the application must run serially on a single CPU core, and thus cannot exploit the native parallelism of Oracle Database.

See Also

Iterative Data Processing: Row-By-Row

4.3.1.4 Iterative Data Processing: Manual Parallelism

Manual parallelism uses the same iterative algorithm as row-by-row and array processing, but enables multiple server processes to work on the job concurrently.

In a typical implementation, the application scans the source data multiple times, and then uses the `ORA_HASH` function to divide the data among the parallel insert processes.

The `ORA_HASH` function computes a hash value for a given expression. The function accepts three arguments:

- *`expr`*, which is typically a column name
- *`max_bucket`*, which specifies the number of hash buckets
- *`seed_value`*, which enables multiple results from the same data (the default is 0)

For example, the following statement divides the sales table into 10 buckets of rows, numbered 0 to 9, and returns the rows from bucket 1:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
SELECT * FROM sales WHERE ORA_HASH(cust_id, 9) = 1;
```

If an application uses `ORA_HASH` in this way, and if n hash buckets exists, then each server process operates on 1/*n* of the data.

Assume the functional requirement is the same as in the row-by-row and array examples: to read scan events from source tables, and then insert them into the `stage1_scan_events` table. The primary differences are as follows:

- The scan events are stored in a mass of flat files. The `ext_scan_events_dets` table describes these flat files. The `ext_scan_events_dets.file_seq_nbr` column stores the numerical primary key, and the `ext_file_name` column stores the file name.
- 32 server processes must run in parallel, with each server process querying a different external table. The 32 external tables are named `ext_scan_events_0` through ext_scan_events_31. However, each server process inserts into the same `stage1_scan_events` table.
- You use PL/SQL to achieve the parallelism by executing 32 threads of the same PL/SQL program, with each thread running simultaneously as a separate job managed by Oracle Scheduler. A job is the combination of a schedule and a program.

The following PL/SQL code, which you execute in SQL*Plus on a separate host from the database server, uses manual parallellism:

```
declare
  sqlstmt varchar2(1024) := q'[
-- BEGIN embedded anonymous block
  cursor c is select s.* from ext_scan_events_${thr} s;
  type t is table of c%rowtype index by binary_integer;
  a t;
  rows binary_integer := 0;
begin
  for r in (select ext_file_name from ext_scan_events_dets where ora_hash(file_seq_nbr,$
{thrs}) = ${thr})
  loop
    execute immediate
      'alter table ext_scan_events_${thr} location' || '(' || r.ext_file_name || ')';
    open c;
    loop
      fetch c bulk collect into a limit ${array_size};
      exit when a.count = 0;
      forall i in 1..a.count
        insert into stage1_scan_events d values a(i);
      commit;
--  demo instrumentation
      rows := rows + a.count; if rows > 1e3 then exit when not
sd_control.p_progress('loading','userdefined',rows); rows := 0; end if;
    end loop;
    close c;
  end loop;
end;
-- END   embedded anonymous block
]';

begin
  sqlstmt := replace(sqlstmt, '${array_size}', to_char(array_size));
  sqlstmt := replace(sqlstmt, '${thr}', thr);
  sqlstmt := replace(sqlstmt, '${thrs}', thrs);
  execute immediate sqlstmt;
end;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

This program has three iterative constructs, from outer to inner:

1. An outer `FOR LOOP` that retrieves names of flat files, and uses DDL to specify the flat file name as the location of an external table
2. A middle `LOOP` statement that fetches groups of rows from a query of the external table.
3. An innermost `FORALL` statement that iterates through each group and inserts the rows

In this sample program, you set `$thrs` to 31 in every job, and set `$thr` to a different value between 0 and 31 in every job. For example, job 1 might have `$thr` set to 0, job 2 might have `$thr` set to 1, and so on.

In the program executed by the first job, with $thr set to 0, the outer `FOR LOOP` iterates through the results of the following query:

```
select ext_file_name
from   ext_scan_events_dets
where  ora_hash(file_seq_nbr,31) = 0
```

The `ORA_HASH` function divides the `ext_scan_events_dets` table into 32 evenly distributed buckets, and then the `SELECT` statement retrieves the file names for bucket 0. For example, the query result set might contain the following file names:

```
/disk1/scan_ev_101
/disk2/scan_ev_003
/disk1/scan_ev_077
...
/disk4/scan_ev_314
```

The middle `LOOP` iterates through the list of file names. For example, the first file name in the result set might be `/disk1/scan_ev_101`. For job 1 the external table is named `ext_scan_events_0`, so the first iteration of the `LOOP` changes the location of this table as follows:

```
alter table ext_scan_events_0 location(/disk1/scan_ev_101);
```

In the innermost `FORALL` statement, the `BULK COLLECT` operator retrieves rows from the `ext_scan_events_0` table into an array, inserts the rows into the `stage1_scan_events` table, and then commits the bulk insert. When the program exits the `FORALL` statement, the program proceeds to the next item in the loop, changes the file location of the external table to `/disk2/scan_ev_003`, and then queries, inserts, and commits rows as in the previous iteration. Job 1 continues processing in this way until all records contained in the flat files corresponding to hash bucket 0 have been inserted in the `stage1_scan_events` table.

While job 1 is executing, the other 31 Oracle Scheduler jobs execute in parallel. For example, job 2 sets `$thr` to 1, which defines the cursor as a query of table `ext_scan_events_1`, and so on through job 32, which sets `$thr` to 31 and defines the cursor as a query of table `ext_scan_events_31`. In this way, each job simultaneously reads a different subset of the scan event files, and inserts the records from its subset into the same `stage1_scan_events` table.

The manual parallelism technique has the following advantages over the alternative iterative techniques:

- It performs far better on large data sets because server processes are working in parallel. For example, if 32 processes are dividing the work, and if the database has sufficient CPU and memory resources and experiences no contention, then the database might perform 32 insert jobs in the time that the array technique took to perform a single job. The performance gain for a large data set is often an order of magnitude greater than serial techniques.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- When the application uses `ORA_HASH` to distribute the workload, each thread of execution can access the same amount of data. If each thread reads and writes the same amount of data, then the parallel processes can finish at the same time, which means that the database utilizes the hardware for as long as the application takes to run.

The manual parallelism technique has the following disadvantages:

- The code is relatively lengthy, complicated, and difficult to understand. The algorithm is complicated because the work of distributing the workload over many threads falls to the developer rather than the database. Effectively, the application runs serial algorithms in parallel rather than running a parallel algorithm.
- Typically, the startup costs of dividing the data have a fixed overhead. The application must perform a certain amount of preparatory work before the database can begin the main work, which is processing the rows in parallel. This startup limitation does not apply to the competing techniques, which do not divide the data.
- If multiple threads perform the same operations on a common set of database objects, then lock and latch contention is possible. For example, if 32 different server processes are attempting to update the same set of buffers, then buffer busy waits are probable. Also, if multiple server processes are issuing `COMMIT` statements at roughly the same time, then log file sync waits are probable.
- Parallel processing consumes significant CPU resources compared to the competing iterative techniques. If the database host does not have sufficient cores available to process the threads simultaneously, then performance suffers. For example, if only 4 cores are available to 32 threads, then the probability of a thread having CPU available at a given time is 1/8.

4.3.2 Set-Based Processing

Set-based processing is a SQL technique that processes a data set inside the database.

In a set-based model, the SQL statement defines the result, and allows the database to determine the most efficient way to obtain it. In contrast, iterative algorithms use conditional logic to pull each row or group of rows from the database to the client application, process the data on the client, and then send the data back to the database. Set-based processing eliminates the network round-trip and database API overhead because the data never leaves the database. It reduces the number of COMMITs.

Assume the same functional requirement as in the previous examples. The following SQL statements meet this requirement using a set-based algorithm:

```
alter session enable parallel dml;
insert /*+ APPEND */ into stage1_scan_events d
  select s.* from ext_scan_events s;
commit;
```

Because the `INSERT` statement contains a subquery of the `ext_scan_events` table, a *single* SQL statement reads and writes all rows. Also, the application executes a *single* `COMMIT` after the database has inserted all rows. In contrast, iterative applications execute a `COMMIT` after the insert of each row or each group of rows.

The set-based technique has significant advantages over iterative techniques:

- As demonstrated in Oracle Real-World Performance demonstrations and classes, the performance on large data sets is orders of magnitude faster. It is not unusual for the run time of a program to drop from several hours to several seconds. The improvement in performance for large data sets is so profound that iterative techniques become extremely difficult to justify.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- A side-effect of the dramatic increase in processing speed is that DBAs can eliminate long-running and error-prone batch jobs, and innovate business processes in real time. For example, instead of running a 6-hour batch job every night, a business can run a 12-seconds job as needed during the day.
- The length of the code is significantly shorter, a short as two or three lines of code, because SQL defines the result and not the access method. This means that the database, rather than the application, decides the best way to divide, retrieve, and manipulate the rows.
- In contrast to manual parallelism, parallel DML is optimized for performance because the database, rather than the application, manages the processes. Thus, it is not necessary to divide the workload manually in the client application, and hope that each process finishes at the same time.
- When joining data sets, the database automatically uses highly efficient hash joins instead of relatively inefficient application-level loops.
- The `APPEND` hint forces a direct-path load, which means that the database creates no redo and undo, thereby avoiding the waste of I/O and CPU. In typical ETL workloads, the buffer cache poses a problem. Modifying data inside the buffer cache, and then writing back the data and its associated undo and redo, consumes significant resources. Because the buffer cache cannot manage blocks fast enough, and because the CPU costs of manipulating blocks into the buffer cache and back out again (usually one 8 K block at a time) are high, both the database writer and server processes must work extremely hard to keep up with the volume of buffers.

The disadvantages of set-based processing:

- The techniques are unfamiliar to many database developers, so they are more difficult. The `INSERT` example is relatively simple. However, more complicated algorithms required more complicated statements that may require multiple outer joins. Developers who are not familiar with pipelining outer joins and using `WITH` clauses and `CASE` statements may be daunted by the prospect of both writing and understanding set-based code.
- Because a set-based model is completely different from an iterative model, changing it requires completely rewriting the source code. In contrast, changing row-by-row code to array-based code is relatively trivial.

Despite the disadvantages of set-based processing, the Oracle Real-World Performance group believes that the enormous performance gains for large data sets justify the effort.

ⓘ Videos

- RWP #7 Set-Based Processing
- RWP #8: Set-Based Parallel Processing
- RWP #9: Set-Based Processing--Data Deduplication
- RWP #10: Set-Based Processing--Data Transformations
- RWP #11: Set-Based Processing--Data Aggregation

Copyright © 1999, 2025, Oracle and/or its affiliates.

5 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

5.1 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 they can use the application. The user has this role only while they are logged in to the application. When the user logs out, the role is revoked.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Database Security Guide* more information about privilege and role authorization
> - *Oracle Database Security Guide* more information about secure application roles

Example 5-1 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 5-1 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;
/
```

5.2 Automating Database Logins

To automate database logins, 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 5-2 shows a simple logon trigger that executes a PL/SQL procedure.

Example 5-2 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

5.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- **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 sales representative who owns the order 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 them from performing normal administrative activities.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 administrators can log in, which computers they can use, whether they 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 AI Database Data Redaction Guide* for more information about Oracle Data Redaction
- *Oracle Label Security Administrator's Guide* for more information about Oracle Label Security
- *Oracle AI Database Vault Administrator's Guide* for more information about Oracle Database Vault

5.4 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

5.4.1 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 5-3 shows how to use the `AUTHID` clause in a `CREATE PROCEDURE` statement to specify definer's rights or invoker's rights.

Example 5-3 Creating Program Units with Definer's Rights or Invoker's Rights

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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

5.4.2 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 5-4 shows how invoking user `hazel` can grant user `harold` the `INHERIT PRIVILEGES` privilege.

Example 5-4 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

5.4.3 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 5-5 shows how to use the `-definer` option in a `loadjava` command.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 5-5 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

5.5 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

5.6 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 12*c* auditing:** For the most part, this option enables you to use the pre-Release 12*c* auditing, in which audit records are written to different locations using their own formats. However, Release 12*c* 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 12*c* 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 12*c* auditing
- *Oracle Database Upgrade Guide* for information about migrating to unified auditing

Copyright © 1999, 2025, Oracle and/or its affiliates.

6

High Availability

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

Topics:

- Transparent Application Failover (TAF)
- Oracle Connection Manager in Traffic Director Mode
- About Fast Application Notification (FAN)
- About Fast Connection Failover (FCF)
- About Application Continuity
- About Transaction Guard
- About Service and Load Management for Database Clouds

6.1 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

6.1.1 About Transparent Application Failover

Transparent Application Failover (TAF) is a client-side feature of Oracle Call Interface (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 during a recovery.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 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`)

6.1.2 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 Also

- About Transparent Application Failover for more information about configuration

6.1.3 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
- Replaying 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Configuring Transparent Application Failover for specific callback registration information for each interface

6.2 Oracle Connection Manager in Traffic Director Mode

This feature allows the Oracle database Connection Manager (CMAN) to be configured in Traffic Director mode to serve clients connecting to different database services, with HA and performance features configurable at the router level, useful to all the clients connected.

Oracle Connection Manager in Traffic Director mode furnishes support for:

- Transparent performance enhancements and connection multiplexing
  - With multiple CMAN in Traffic Director mode instances, applications get increased scalability through client-side connection-time load balancing or with a load balancer (BIG-IP, NGINX, and others)
- Zero application downtime including: planned database maintenance or pluggable database (PDB) relocation and unplanned database outages for read-mostly workloads.
- High Availability of CMAN in Traffic Director mode to avoid a single point of failure.
- Security and isolation:
  - Database Proxy supporting transmission control protocol/transmission control protocol secure (TCP/TCPS) and protocol conversion
  - Firewall based on the IP address, service name, and secure socket layer/transport layer security (SSL/TLS) wallets
  - Tenant isolation in a multi-tenant environment
  - Protection against denial-of-service and fuzzing attacks
  - Secure tunneling of database traffic across Oracle Database On-premises and Oracle Cloud

Related Topics

- *Oracle AI Database Security Guide*
- *Oracle AI Database Net Services Administrator's Guide*
- *Oracle AI Database Net Services Administrator's Guide*
- *Oracle AI Database Net Services Administrator's Guide*

6.3 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 one of the following integrated Oracle clients:

- OCI session pools
- Universal Connection Pool for Java
- Thin JDBC Driver (12.2 and later)
- ODP.NET managed and un-managed providers
- All WebLogic server data sources, and Oracle Tuxedo

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. 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 12*c* Release 1 (12.1), ONS is the primary notification mechanism for a new client (Oracle Database 12*c* Release 1 (12.1)) and a new server (Oracle Database 12*c* Release 1 (12.1)), while the AQ HA notification feature is deprecated and maintained only or backward compatibility when there is an older OCI or ODP.NET unmanaged client (pre-Oracle Database 12*c* Release 1 (12.1)) or old server (pre-Oracle Database 12*c* Release 1 (12.1)).

When you use JDBC or Oracle Database 12 *c* Release 1 (12.1.0.1) OCI or ODP.NET clients, the Oracle Notification Service is automatically configured using your TNS. When you use OCI-based clients, set HA notifications (`-notification = TRUE`) for your services and set EVENTS in `oraccess.xml`.

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

6.3.1 About Receiving FAN Event Notifications

Copyright © 1999, 2025, Oracle and/or its affiliates.

Starting from Oracle Database 12c Release 2 (12.2), the Oracle RAC FAN APIs provide an alternative for taking advantage of the high-availability (HA) features of Oracle Database, if you do not use Universal Connection Pool or Oracle WebLogic Server with Active Grid Link (AGL). This feature depends on the Oracle Notification System (ONS) message transport mechanism.

This feature requires configuring your system, servers, and clients to use ONS. For using Oracle RAC Fast Application Notification, the `simplefan.jar` file must be present in the `CLASSPATH`, and either the `ons.jar` file must be present in the `CLASSPATH` or an Oracle Notification Services (ONS) client must be installed and running in the client system.

See Also

Oracle AI Database JDBC Developer's Guide for more information about Oracle RAC FAN APIs.

6.4 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 12*c* Release 1 (12.1), ONS is the primary notification mechanism for a new client (Oracle Database 12*c* Release 1 (12.1)) and a new server (Oracle Database 12*c* 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 12*c* Release 1 (12.1)) or old server (pre-Oracle Database 12*c* 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 Universal Connection Pool (UCP) 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

6.5 About Application Continuity

Application Continuity masks planned or unplanned outages (that cause database session unavailability) by attempting to rebuild the database sessions transactional and non-transactional states, so the outage appears to the user as nothing more than a delayed execution.

Transparent Application Continuity (TAC) transparently records the database session and transactional state so the database can be recovered following recoverable outages. This is done safely and with no reliance on application knowledge or application code changes, thus allowing TAC to be enabled as a standard. TAC is enabled by default starting Oracle Database Release 21c.

Oracle Database Release 21c introduces the `RESET_STATE` service attribute, which you can use to reset state in a session. You can set the `RESET_STATE` service attribute in the `DBMS_SERVICE.CREATE_SERVICE` procedure. It is executed at the end of a request before the next processing occurs in that request. You can use this service attribute when a session is returned to a connection pool, so that the session state does not leak from one session usage to the next.

The `RESET_STATE` attribute can have the following values:

- `NONE`: If you set the `RESET_STATE` attribute to `NONE`, then the session is not cleaned at the end of the request. This is the default value of this attribute.
- `LEVEL1`: If you set the `RESET_STATE` attribute to `LEVEL1`, then the session states, which cannot be restored, are reset.

See the `RESET_STATE` topic below for more information.

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, within an Active Data Guard, Multitenant using PDB relocate with Oracle RAC or across RACs or across to Active Data Guard (ADG).

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Real Application Clusters Administration and Deployment Guide* for more information about Application Continuity
> - CREATE_SERVICE Procedure in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `CREATE_SERVICE` and other `DBMS_SERVICE` subprograms

6.5.1 Reset Database Session State to Prevent Application State Leaks, Use RESET_STATE

When you use the `RESET_STATE` service attribute, the session state set in a request by an application is cleared when the request ends.

The `RESET_STATE` service attribute (`RESET_STATE`) is recommended for all stateless applications to prevent the leakage of a session state to later reuses.

> **Note**
>
> As of Oracle AI Database 26ai, `RESET_STATE` works independent of Application Continuity.

Stateless Applications

A stateless application is an application program that does not use the session state of one request, such as context and PL/SQL states that were set by a prior usage of that session, in another web request or similar connection pool usage. The necessary state to handle the request is contained within the request, as a part of the request itself; in the URL, query-string parameters, request body, or headers.

In a cloud environment, it is preferable for applications to be stateless for scalability and portability. Being stateless enables greater scalability because the server does not have to maintain, update, or communicate a session state. For example, load balancers do not have to consider session affinity for stateless systems. Most modern web applications are stateless.

Prevent Application State Leaking to Later Usages

Setting the session state in a request leaves the session with the current state, meaning that subsequent usages of that session can see the current session state. For example, when an application borrows and returns a connection to a connection pool, if the sessions state is not cleared, the next usage of that connection can see the session state of the previous usage.

`RESET_STATE` is an attribute of the database service. When you use `RESET_STATE`, the session state set by an application in a request is cleared when the database request ends. When `RESET_STATE` is used, an application can depend on the state being reset at the end of a request. Without `RESET_STATE`, application developers must cancel their cursors and clear the session state that has been set before returning their connections to a pool for reuse.

`RESET_STATE` is used with applications that are stateless between requests. These types of applications use the session state in a request, and do not rely on that session state in the later requests. The necessary session state that the request needs is contained within the request

Copyright © 1999, 2025, Oracle and/or its affiliates.

itself. REST, Oracle REST Data Services (ORDS), Oracle Application Express (APEX) are examples of stateless applications.

RESET_STATE is available for all applications that use Oracle or third-party connection pools with request boundaries and is supported for all Oracle clients, such as Oracle Call Interface (OCI) session pools or Java Database Connectivity (JDBC) universal connection pools. Setting RESET_STATE enables automatic resetting of session states at an explicit end of request. RESET_STATE does not apply to implicit request boundaries, such as those with DRCP implicit statement caching or Transparent Application Continuity (TAC).

OPTIONS FOR RESET_STATE

RESET_STATE = LEVEL1

When you use RESET_STATE at LEVEL1, the database clears complex session states when each request ends.

Using RESET_STATE LEVEL1 has the following impact at the end of a request:

- Cursors are canceled.
- PL/SQL global variables are cleared.
- Temporary tables that have a session-based duration are truncated.
- Temporary LOBs that have a session-based duration are cleared.
- Session local sequences are reset.
- DBMS_OUTPUT buffer is cleared.

The order of processing for resetting session states at LEVEL1 is the following:

1. Borrow from a connection pool or other request explicitly marked
2. Label Callback
3. Begin Request
4. User request runs
5. End Request
6. RESET_STATE LEVEL1 is applied automatically by the server

RESET_STATE is a very important database feature because developers can rely on these listed session states being reset when a session is returned to a connection pool with request boundaries. This can be an Oracle connection pool or a custom connection pool with added request boundaries.

RESET_STATE also improves your protection when using TAC, because the session state is clean at the beginning of a new request.

6.6 About Transaction Guard

Transaction Guard is a reliable protocol and interface that returns the commit outcome of the current in-flight transaction when an error, or a time-out is returned to the client. Applications can leverage the Transaction Guard interface to code graceful recoverable error handling. Providing unambiguous message during an outage greatly improves the user experience.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

identifies the database transaction from the application perspective. Applications use the Transaction Guard interface to obtain a known commit 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 non-transactional states - see `USER_CALL_COMPLETED`.

See Also

Using Transaction Guard

6.7 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 12*c* 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 12*c* 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 12*c* 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
- Introduction to Global Data Services in *Oracle Database Global Data Services Concepts and Administration Guide* for more information about global service management in a database cloud
- Installation and Configuration in *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.
- *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

Copyright © 1999, 2025, Oracle and/or its affiliates.

7

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**
>
> - PL/SQL for Application Developers
> - *Oracle AI Database PL/SQL Language Reference* for a complete description of PL/SQL

7.1 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.

> **See Also**
>
> PL/SQL Data Types

7.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

PL/SQL Dynamic SQL

7.3 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`.

See Also

- *Oracle Database Reference*
- PLSQL_OPTIMIZE_LEVEL Compilation Parameter

7.4 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.

See Also

Compiling PL/SQL Units for Native Execution for more information about compiling PL/SQL units for native execution

7.5 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.

See Also

Oracle Database PL/SQL Language Reference

7.6 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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, which propagates changes to dependent objects.
- Conditional compilation is subject to restrictions.

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

7.7 Bulk Binding

Bulk binding minimizes the performance overhead of the communication between PL/SQL and SQL, which can greatly improve performance.

> **See Also**
>
> Overview of Bulk Binding.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

- SQL Processing for Application Developers
- Using SQL Data Types in Database Applications
- Registering Application Data Usage with the Database
- Using Regular Expressions in Database Applications
- Using Indexes in Database Applications
- Maintaining Data Integrity in Database Applications

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

8
SQL Processing for Application Developers

This chapter explains what application developers must know about how Oracle Database processes SQL statements.

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
- Schema Annotations
- Using `IF EXISTS` and `IF NOT EXISTS`

> **See Also**
>
> *Oracle Database Concepts*

8.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.1.1 Stages of SQL Statement Processing

> **Note**
>
> DML statements use all stages. Transaction management, session management, and system management SQL statements use only `stage 2` and `stage 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.

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.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

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.

> **See Also**
>
> - *Oracle Database Concepts* for information about parsing
> - Shared SQL Areas
> - *Oracle Database Concepts* for information about the `DEFINE` stage
> - *Oracle Call Interface Developer's Guide*
> - *Pro*C/C++ Programmer's Guide*

8.1.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 AI Database SQL Tuning Guide* for more information about shared SQL

8.2 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

8.2.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.2.2 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.
  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.
- 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.

  To collect statistics unrelated to the cost-based optimizer (such as information about free list blocks), use the SQL statement `ANALYZE`.
- 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.
- Increase user productivity and query efficiency by including user-written PL/SQL functions in SQL expressions. For details, see.

See Also

- *Oracle Database Concepts* for more information about transaction management
- PL/SQL for Application Developers
- Developing Applications with Multiple Programming Languages
- *Oracle Database SQL Language Reference* for more information about hints and `ANALYZE` statement
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_STATS` package and `DBMS_APPLICATION_INFO` package
- Invoking Stored PL/SQL Functions from SQL Statements

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.2.3 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 8-1 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 8-1 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 or the appropriate initialization parameter.

> ⓘ **Note**
>
> You cannot change the default `IMMEDIATE` and `WAIT` action for distributed transactions.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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_WRITENOWAIT`
- `OCI_TRANS_WRITEBATCH`
- `OCI_TRANS_WRITEIMMED`

> **Caution**
>
> `OCI_TRANS_WRITENOWAIT` 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.

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.

> **See Also**
>
> - *Oracle Call Interface Programmer's Guide* for information about the `OCITransCommit` function
> - *Oracle Database SQL Language Reference*
> - *Oracle Database Reference* for information about initialization parameters

8.2.4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

8.2.4.1 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`.

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.

See Also

Understanding DBMS_APP_CONT.GET_LTXID_OUTCOME

8.2.4.1.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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).
  - When not using Real Application Clusters, use the `DBMS_SERVICE` package.

  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

- Transaction Guard Coverage, for a list of the sources whose commits Transaction Guard supports
- Transaction Guard Exclusions, for a list of the sources whose commits Transaction Guard does not support
- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about SRVCTL
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_SERVICE` package

8.2.4.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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`

8.2.4.2.1 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.

> **See Also**
>
> - *Oracle AI Database JDBC Developer's Guide*
> - *Oracle Call Interface Developer's Guide*
> - *Oracle Data Provider for .NET Developer's Guide*

8.2.4.2.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

8.2.4.2.3 USER_CALL_COMPLETED Parameter

Some transactions return information upon completion. For example: A transaction that uses commit on success (auto-commit) might returns 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.

8.2.4.2.4 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 DBMS_APP_CONT.GET_LTXID_OUTCOME identifies a transaction that is older than the in-flight transaction. Your application must get the LTXID immediately before passing it to DBMS_APP_CONT.GET_LTXID_OUTCOME.
ORA-14951 - CLIENT_AHEAD	The client is ahead of the server. Either the server was "flashed back" to an earlier state, was recovered using media recovery, or is a standby database that was opened earlier and has lost data.
ORA-14906 - SAME_SESSION	Executing GET_LTXID_OUTCOME is not supported on the session that owns the LTXID, because the execution would block further processing on that session.
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 terminated 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- Using Oracle Flashback Technology
- *Oracle Data Guard Concepts and Administration* for information about media recovery and standby databases

8.2.4.3 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:

   a. If necessary, clean up state changes on the client side.

   b. 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.

8.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

8.4 Locking Tables Explicitly

Oracle Database has default locking mechanisms that ensure data concurrency, data integrity, and statement-level read consistency. 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).

To override default locking at the transaction level, use any of these SQL statements:

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

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

The initialization parameter `DML_LOCKS` 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.

> **Note**
>
> 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.

> **See Also**
>
> - *Oracle AI Database Concepts*
> - Ensuring Repeatable Reads with Read-Only Transactions
> - Using Serializable Transactions for Concurrency Control
> - *Oracle AI Database SQL Language Reference*

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

8.4.1 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.

8.4.2 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

You can specify several tables or views to lock in the same mode; however, only a single lock mode can be specified for each `LOCK TABLE` statement.

> **Note**
>
> When a table is locked, all rows of the table are locked. No other user can modify the table.

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

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

  You acquire the table lock only if it is immediately available; otherwise, an error notifies you that the lock is 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.

> **See Also**
>
> - Explicitly Acquiring Row Locks
> - *Oracle Database SQL Language Reference*

Topics:

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

8.4.2.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.2.2 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 8-1. Because the departments table is rarely updated, locking it probably does not cause many other transactions to wait long.

Example 8-1 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
COMMIT;  -- COMMIT releases lock
```

8.4.2.3 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.

8.4.2.4 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.

8.4.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.4 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.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.5 Examples of Concurrency Under Explicit Locking

Table 8-2 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.

Table 8-2 Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2
1	**LOCK TABLE hr.departments IN ROW SHARE MODE;** Statement processed.	
2		**DROP TABLE hr.departments;** DROP TABLE hr.departments * ORA-00054 (Exclusive DDL lock not possible because Transaction 1 has table locked.)
3		**LOCK TABLE hr.departments IN EXCLUSIVE MODE NOWAIT;** ORA-00054
4		**SELECT location_id FROM hr.departments WHERE department_id = 20 FOR UPDATE OF location_id;** LOCATION_ID ----------- DALLAS 1 row selected.
5	**UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;** (Waits because Transaction 2 locked same rows.)	

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2
6		**ROLLBACK;** (Releases row locks.)
7	1 row processed. **ROLLBACK;**	
8	**LOCK TABLE hr.departments** **IN ROW EXCLUSIVE MODE;** Statement processed.	
9		**LOCK TABLE hr.departments** **IN EXCLUSIVE MODE** **NOWAIT;** ORA-00054
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;**
14	**SELECT location_id** **FROM hr.departments** **WHERE department_id = 20** **FOR UPDATE OF location_id;** LOCATION_ID ----------- DALLAS 1 row selected.	

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2
15		**UPDATE hr.departments** **SET location_id = 'NEW YORK'** **WHERE department_id = 20;** 1 row processed. (Waits because Transaction 1 locked same rows.)
16	**ROLLBACK;**	
17		1 row processed. (Conflicting locks were released.) **ROLLBACK;**
18	**LOCK TABLE hr.departments** **IN ROW SHARE MODE** Statement processed.	
19		**LOCK TABLE hr.departments** **IN EXCLUSIVE MODE** **NOWAIT;** ORA-00054
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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2
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.)
25	**ROLLBACK;**	
26		1 row processed. (Conflicting table lock released.) **ROLLBACK;**
27	**LOCK TABLE hr.departments** **IN SHARE ROW EXCLUSIVE MODE;** Statement processed.	
28		**LOCK TABLE hr.departments** **IN EXCLUSIVE MODE** **NOWAIT;** ORA-00054
29		**LOCK TABLE hr.departments** **IN SHARE ROW EXCLUSIVE MODE** **NOWAIT;** ORA-00054
30		**LOCK TABLE hr.departments** **IN SHARE MODE** **NOWAIT;** ORA-00054

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2

31		`LOCK TABLE hr.departments`
		`IN ROW EXCLUSIVE MODE`
		`NOWAIT;`
		
		`ORA-00054`

32		`LOCK TABLE hr.departments`
		`IN SHARE MODE`
		`NOWAIT;`
		
		`ORA-00054`

33		`SELECT location_id`
		`FROM hr.departments`
		`WHERE department_id = 20;`
		
		`LOCATION_ID`
		`-----------`
		`DALLAS`
		
		`1 row selected.`

34		`SELECT location_id`
		`FROM hr.departments`
		`WHERE department_id = 20`
		`FOR UPDATE OF location_id;`
		
		`LOCATION_ID`
		`-----------`
		`DALLAS`
		
		`1 row selected.`

35		`UPDATE hr.departments`
		`SET location_id = 'NEW YORK'`
		`WHERE department_id = 20;`
		
		(Waits because Transaction 1 has conflicting table lock.)

36	`UPDATE hr.departments`
	`SET location_id = 'NEW YORK'`
	`WHERE department_id = 20;`
	
	(Waits because Transaction 2 locked same rows.)
		(Deadlock.)

37	`Cancel operation.`
	
	`ROLLBACK;`	

38		`1 row processed.`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2
39	`LOCK TABLE hr.departments IN EXCLUSIVE MODE;`	

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.)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2

47	`UPDATE hr.departments`
	`SET department_id = 30`
	`WHERE department_id = 20;`
	
	`1 row processed.`	

48	`COMMIT;`	
49		`0 rows selected.` (Transaction 1 released conflicting lock.)
50	`SET TRANSACTION READ ONLY;`	

51	`SELECT location_id`
	`FROM hr.departments`
	`WHERE department_id = 10;`
	
	`LOCATION_ID`
	`-----------`
	`BOSTON`	

52		`UPDATE hr.departments`
		`SET location_id = 'NEW YORK'`
		`WHERE department_id = 10;`
		
		`1 row processed.`

53	`SELECT location_id`
	`FROM hr.departments`
	`WHERE department_id = 10;`
	
	`LOCATION_ID`
	`-----------`
	`BOSTON`
	
	(Transaction 1 does not see uncommitted data.)	

54		`COMMIT;`

55	`SELECT location_id`
	`FROM hr.departments`
	`WHERE department_id = 10;`
	
	`LOCATION_ID`
	`-----------`
	`BOSTON`
	
	(Same result even after Transaction 2 commits.)	

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-2 (Cont.) Examples of Concurrency Under Explicit Locking

Time Point	Transaction 1	Transaction 2
56	`COMMIT;`	

57	`SELECT location_id`
	`FROM hr.departments`
	`WHERE department_id = 10;`
	
	`LOCATION_ID`
	`-----------`
	`NEW YORK`
	
	(Sees committed data.)	

> **See Also**
>
> *Oracle AI Database Concepts*

8.5 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

8.5.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 8-2 shows how the Pro*COBOL precompiler uses locks to ensure that there are no conflicts when multiple people must access a single device.

Example 8-2 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.
```

8.5.2 Viewing and Monitoring Locks

Table 8-3 describes the Oracle Database facilities that display locking information for ongoing transactions within an instance.

Table 8-3 Ways to Display Locking Information

Tool	Description
Performance Monitoring Data Dictionary Views	See *Oracle Database Administrator's Guide*.
`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`.)

8.6 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

`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

8.6.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Transaction Interaction	Definition
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 8-4 shows what kind of interactions are possible at each isolation level.

Table 8-4 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 8-5 shows which ANSI/ISO SQL transaction isolation levels Oracle Database provides.

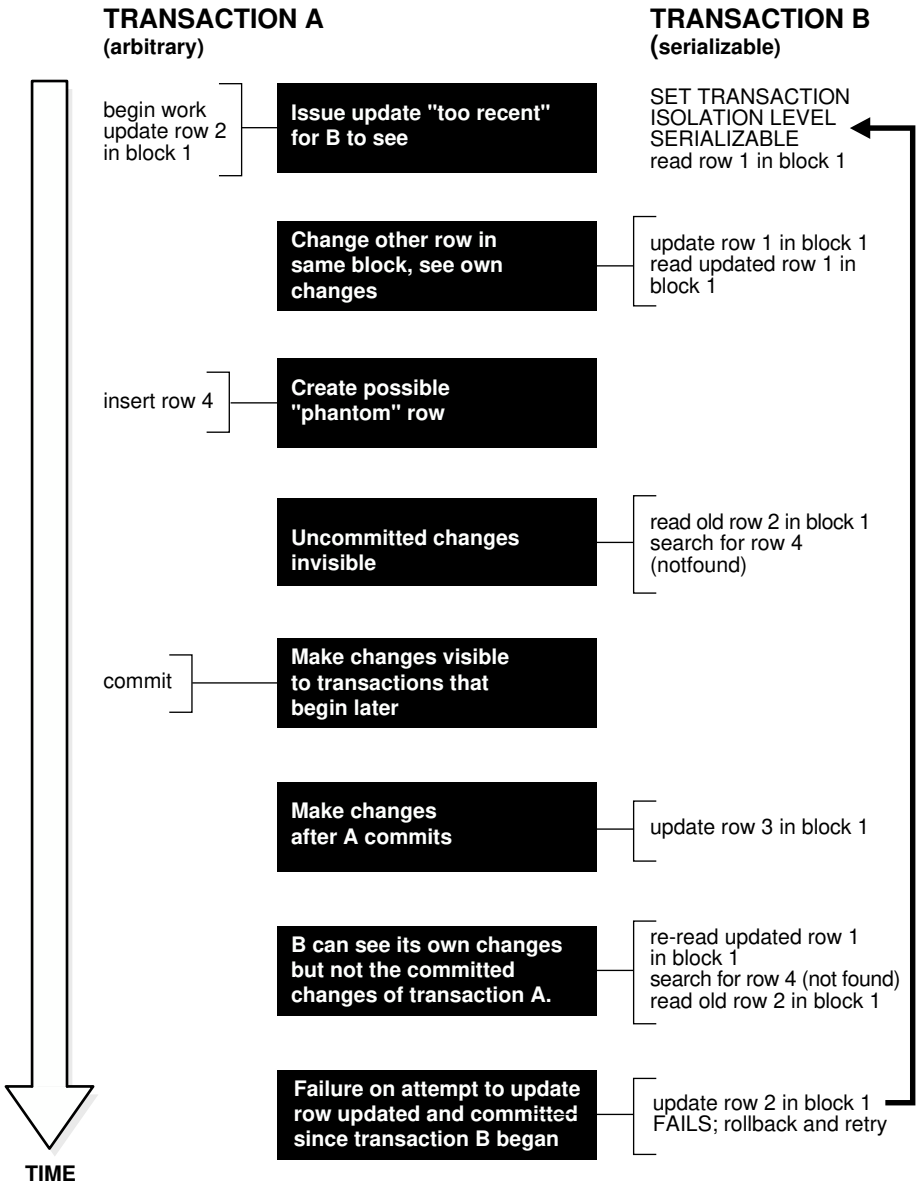
Table 8-5 ANSI/ISO SQL Isolation Levels Provided by Oracle Database

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.
`REPEATABLE READ`	Yes, if you set the transaction isolation level to `SERIALIZABLE`.
`SERIALIZABLE`	Yes, if you set the transaction isolation level to `SERIALIZABLE`.

Figure 8-1 shows how an arbitrary transaction (that is, one that is either `SERIALIZABLE` or `READ COMMITTED`) interacts with a serializable transaction.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 8-1 Interaction Between Serializable Transaction and Another Transaction



8.6.2 Setting Isolation Levels

To set the transaction isolation level for every transaction in your session, use the `ALTER SESSION` statement.

To set the transaction isolation level for a specific transaction, use the `ISOLATION LEVEL` clause of the `SET TRANSACTION` statement. The `SET TRANSACTION` statement, must be the first statement in the transaction.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> If you set the transaction isolation level to `SERIALIZABLE`, then you must use the `ALTER TABLE` statement to set the `INITRANS` parameter to at least 3. Use higher values for tables for which many transactions update the same blocks. For more information about `INITRANS`.

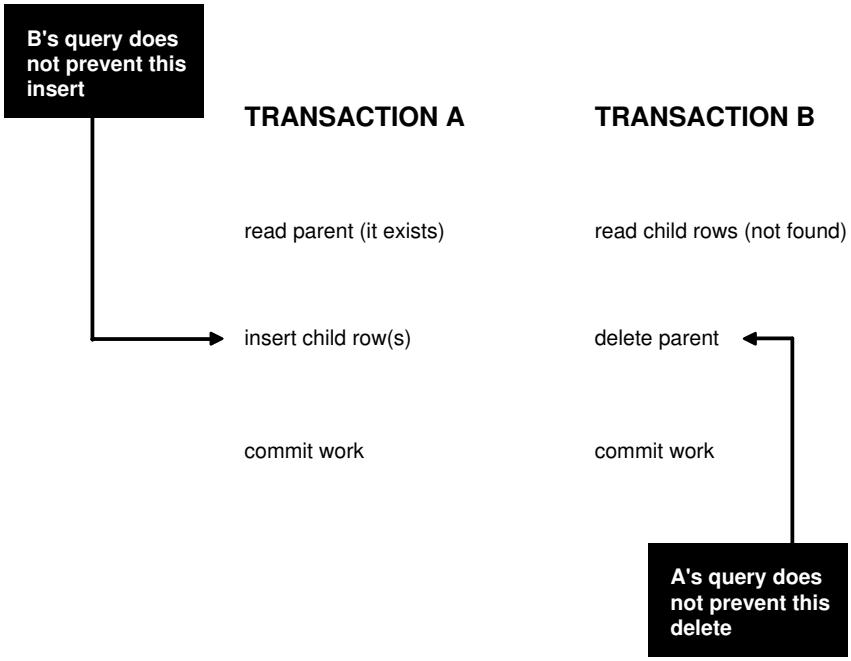
> **See Also**
>
> *Oracle Database SQL Language Reference*

8.6.3 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 8-2, 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 8-2 Referential Integrity Check



Copyright © 1999, 2025, Oracle and/or its affiliates.

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.
- 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.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database SQL Language Reference* for information about the `FOR UPDATE` clause of the `SELECT` statement
- *Oracle Database PL/SQL Language Reference* for more information about using triggers to maintain referential integrity between parent and child tables

8.6.4 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

8.6.4.1 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

8.6.4.1.1 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.

8.6.4.1.2 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 primary table with a detail table can see a primary 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

8.6.4.2 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 Serializable Transactions and Referential Integrity reads do not block writes in either READ COMMITTED or SERIALIZABLE transactions.

Table 8-6 summarizes the similarities and differences between READ COMMITTED and SERIALIZABLE transactions.

Table 8-6 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

Serializable Transactions and Referential Integrity

8.7 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_TIMEOUT` 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).

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.

Caution

Do not issue a nonblocking DDL statement in an autonomous transaction. See Autonomous Transactions for information about autonomous transactions

See Also

- *Oracle Database Reference* for information about the `DDL_LOCK_TIMEOUT` parameter
- B Automatic and Manual Locking Mechanisms During SQL Operations for a list of nonblocking DDL statements
- ALTER DATABASE for information about enabling and disabling supplemental logging

8.8 Autonomous Transactions

Caution

Do not issue a nonblocking DDL statement in an autonomous transaction.

An **autonomous transaction** (AT) is an independent transaction started by another transaction, the **main transaction** (MT). An autonomous transaction lets you suspend the

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

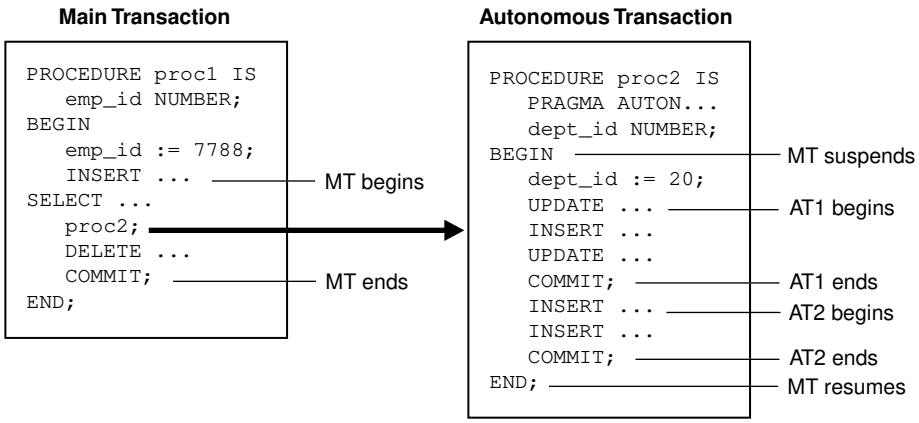
An autonomous transaction runs within an **autonomous scope**; that is, within the scope of an **autonomous routine**—a routine that you mark with the `AUTONOMOUS_TRANSACTION` pragma. In this context, a **routine** is one of these:

- 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 8-3 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 8-3 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 8-3 shows, when one transaction ends, the next SQL statement begins another transaction.

More characteristics of autonomous transactions:

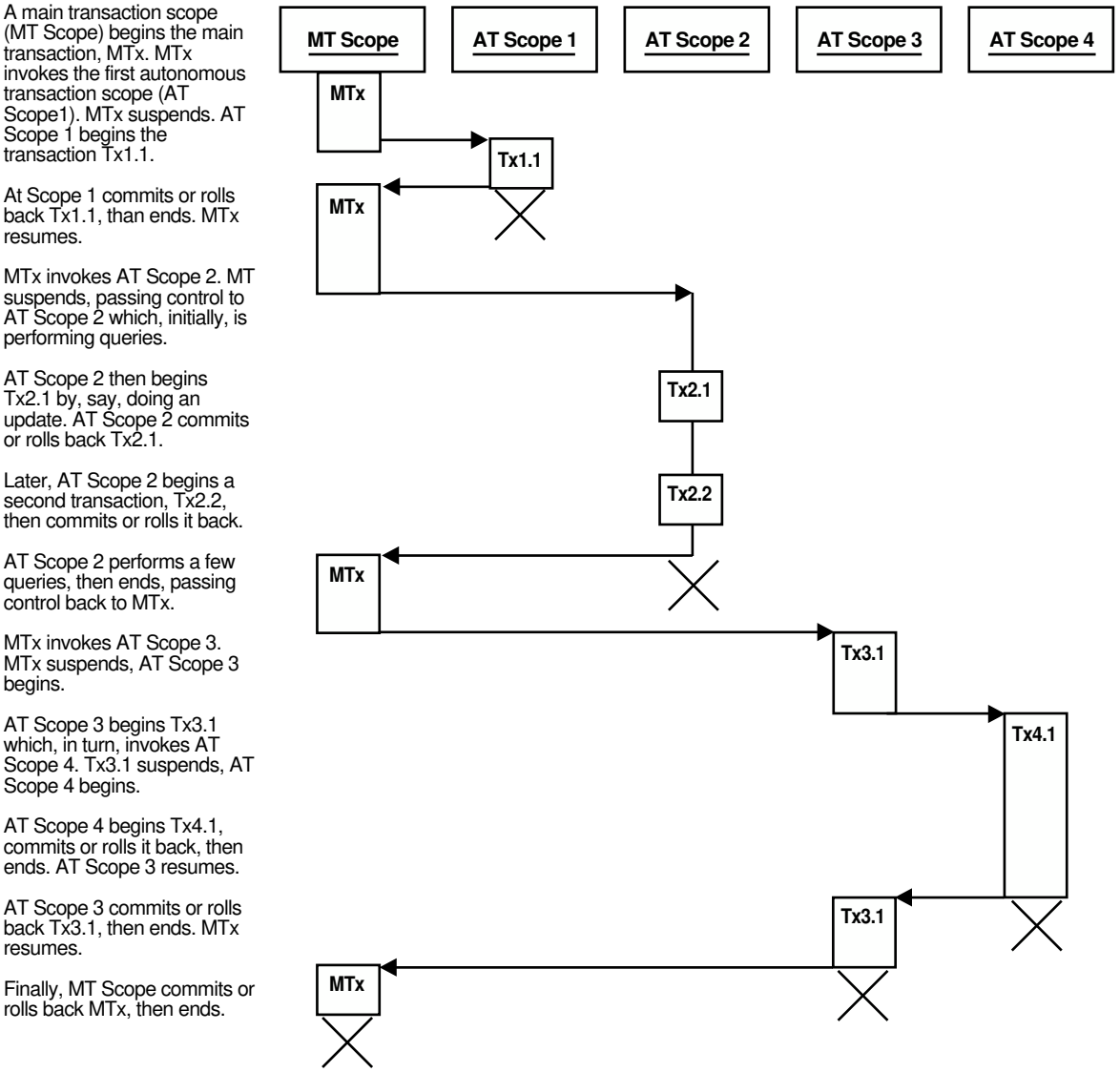
- The changes an autonomous transaction effects do not depend on the state or the eventual disposition of the main transaction. For example:
  - An autonomous transaction does not see changes made by the main transaction.
  - When an autonomous transaction commits or rolls back, it does not affect the 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Autonomous transactions can start other autonomous transactions.

Figure 8-4 shows some possible sequences that autonomous transactions can follow.

Figure 8-4 Possible Sequences of Autonomous Transactions



Topics:

- Examples of Autonomous Transactions
- Declaring Autonomous Routines

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database PL/SQL Language Reference* for detailed information about autonomous transactions
- Nonblocking and Blocking DDL Statements for information about nonblocking DDL statements

8.8.1 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 8-7). There is no dependency between the outcome of an autonomous transaction and that of a main transaction.

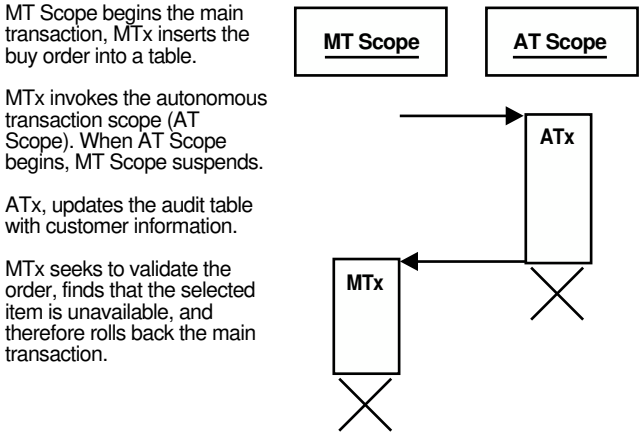
Table 8-7 Possible Transaction Outcomes

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

8.8.1.1 Ordering a Product

Figure 8-5 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 8-5 Example: A Buy Order



Copyright © 1999, 2025, Oracle and/or its affiliates.

8.8.1.2 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).

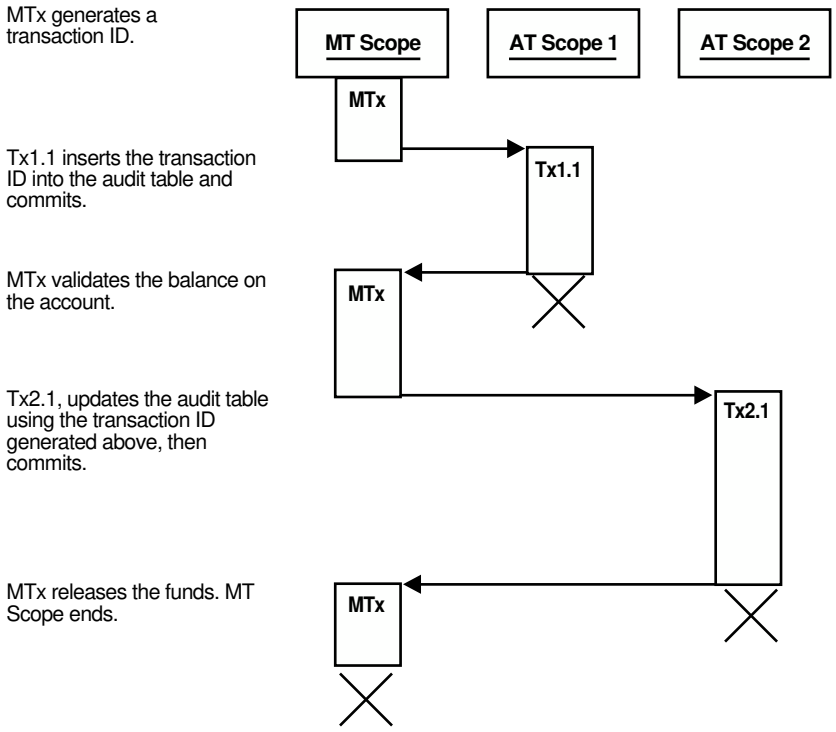
The possible scenarios for this transaction are:

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

8.8.1.2.1 Scenario 1: Sufficient Funds

There are sufficient funds to cover the withdrawal, so the bank releases the funds (see Figure 8-6).

Figure 8-6 Bank Withdrawal—Sufficient Funds

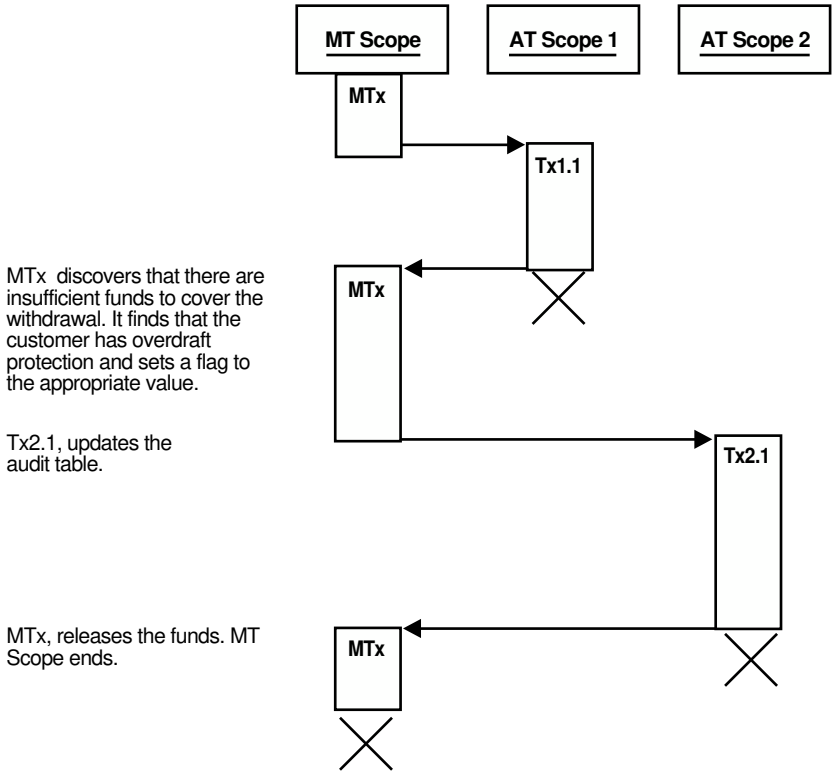


8.8.1.2.2 Scenario 2: Insufficient Funds with Overdraft Protection

There are insufficient funds to cover the withdrawal, but the customer has overdraft protection, so the bank releases the funds (see Figure 8-7).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 8-7 Bank Withdrawal—Insufficient Funds with Overdraft Protection

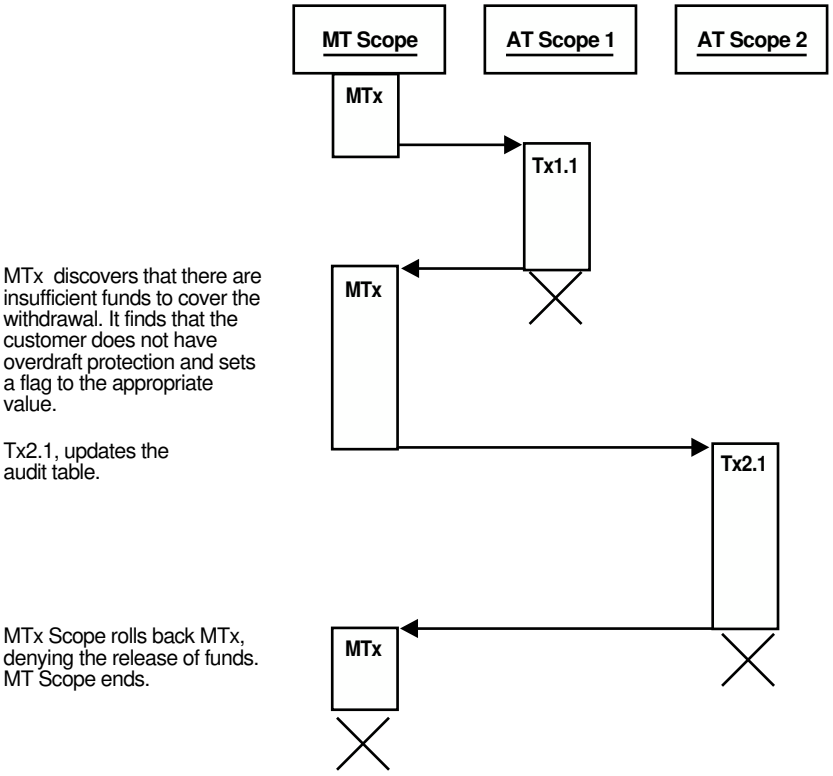


8.8.1.2.3 Scenario 3: Insufficient Funds Without Overdraft Protection

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 8-8).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 8-8 Bank Withdrawal—Insufficient Funds Without Overdraft Protection



8.8.2 Declaring Autonomous Routines

To declare an autonomous routine, use `PRAGMA AUTONOMOUS_TRANSACTION`, which instructs the PL/SQL compiler to mark the routine as autonomous.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about `PRAGMA AUTONOMOUS_TRANSACTION`

In Example 8-3, the function `balance` is autonomous.

Example 8-3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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;
/
```

8.9 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

8.9.1 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 `OCIStmtExecute`.

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.

8.9.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

8.9.2.1 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. 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.

The trigger in Example 8-4 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.

> **See Also**
>
> - *Oracle Database PL/SQL Packages and Types Reference*
> - *Oracle Database PL/SQL Language Reference*

Example 8-4 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  table_space_name  VARCHAR2(64);
  object_name       VARCHAR2(64);
  sub_object_name   VARCHAR2(64);
  msg_body          VARCHAR2(64);
  ret_value         BOOLEAN;
  error_txt         VARCHAR2(64);
  mail_conn         UTL_SMTP.CONNECTION;
BEGIN
 SELECT DISTINCT(SID) INTO cur_sid FROM V$MYSTAT;
 cur_inst := USERENV('instance');
 ret_value := DBMS_RESUMABLE.SPACE_ERROR_INFO
              (err_type,
              object_owner,
              object_type,
              table_space_name,
              object_name,
              sub_object_name);
 IF object_type = 'ROLLBACK SEGMENT' THEN
   INSERT INTO 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;
/
```

8.9.2.2 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` .

The DBA can get additional information from the dynamic performance view `V$_SESSION_WAIT`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- DBA_RESUMABLE
- V$_SESSION_WAIT

8.10 Using IF EXISTS and IF NOT EXISTS

This section explains how to use IF EXISTS and IF NOT EXISTS with CREATE, ALTER, and DROP commands for different object types.

To ensure that your DDL statements are idempotent, the CREATE, ALTER, and DROP commands support the IF EXISTS and IF NOT EXISTS clauses. You can use these clauses to check if a given object exists or does not exist, and ensure that if the check fails, the command is ignored and does not generate an error. The CREATE DDL statement works with IF NOT EXISTS to suppress an error when an object already exists. Similarly, the ALTER and DROP DDL statements support the IF EXISTS clause to suppress an error when an object does not exist.

For example, you can control whether you need to know that a table exists before issuing the DROP command. If the existence of the table is not important, you can pass a statement similar to the following:

```
DROP TABLE IF EXISTS <table_name>...
```

If the table exists, the table is dropped. If the table does not exist, the statement is ignored and hence no error is raised. The same check mechanism exists for the creation of objects.

Assume a scenario where you do not have the IF NOT EXISTS support. Before issuing a query, you expect a table to exist but if it does not exist, a new one must be created. You would leverage PL/SQL or query the data dictionary to know if the table is present. If the table is not present, you would execute a dynamic SQL (EXECUTE IMMEDIATE) to create the table. With the IF NOT EXISTS support, you can issue a CREATE TABLE IF NOT EXISTS <table_name>... command to create a table if the table does not exist. If a table with this name exists, irrespective of the table's structure, the statement is ignored without generating an error.

Topics:

- Using IF NOT EXISTS with CREATE Command
- Using IF EXISTS with ALTER Command
- Using IF EXISTS with DROP Command
- Supported Object Types
- Limitations for CREATE OR REPLACE Statements
- SQL*Plus Output Messages for DDL Statements

8.10.1 Using IF NOT EXISTS with CREATE Command

The IF NOT EXISTS clause when used with the CREATE command, enables you to suppress an error if a given object already exists. If the object already exists, the CREATE command with IF NOT EXISTS suppresses the error, and no error is returned to the user. If the object does not exist, the CREATE command creates a new object.

The CREATE command supports the IF NOT EXISTS clause for many objects types.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Supported Object Types for a complete list of the object types that support the `IF NOT EXISTS` clause.

Here is the syntax and an example of using `IF NOT EXISTS` with the `CREATE` command:

```
CREATE <object type> [IF NOT EXISTS] <rest of syntax>

-- create table if not exists
CREATE TABLE IF NOT EXISTS t1 (c1 number);
```

8.10.2 Using `IF EXISTS` with `ALTER` Command

The `IF EXISTS` clause when used with the `ALTER` command, enables you to suppress an error if a given object does not exist. If the object exists, the `ALTER` command with the `IF EXISTS` clause executes successfully.

The `ALTER` command supports the `IF EXISTS` clause for many objects types.

> **See Also**
>
> Supported Object Types for a complete list of the object types that support the `IF EXISTS` clause.

Here is the syntax and an example of using `IF EXISTS` with the `ALTER` command:

```
ALTER <object type> [IF EXISTS] <rest of syntax>

-- alter table if exists
ALTER TABLE IF EXISTS t1 ADD c2 number;;
```

8.10.3 Using `IF EXISTS` with `DROP` Command

The `IF EXISTS` clause when used with the `DROP` command, enables you to suppress an error if a given object does not exist. If the object exists, the `DROP` command with the `IF EXISTS` clause executes successfully.

The `DROP` command supports the `IF EXISTS` clause for many objects types.

> **See Also**
>
> Supported Object Types for a complete list of the object types that support the `IF EXISTS` clause.

Here is the syntax and an example of using `IF EXISTS` with the `DROP` command:

```
DROP <object type> [IF EXISTS] <rest of syntax>

-- drop table if exists
DROP TABLE IF EXISTS t1;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

8.10.4 Supported Object Types

These are the object types that support the `IF EXISTS` and `IF NOT EXISTS` clauses.

The following object types are supported for `CREATE ... IF NOT EXISTS`, `ALTER ... IF EXISTS`, and `DROP ... IF EXISTS` DDL statements.

Table 8-8 Object Types Supported for `CREATE`, `ALTER`, and `DROP` Commands

CREATE and DROP Commands	ALTER Command
ANALYTIC VIEW	ANALYTIC VIEW
ASSEMBLY	
ATTRIBUTE DIMENSION	ATTRIBUTE DIMENSION
CLUSTER	CLUSTER
DATABASE LINK	DATABASE LINK
DIRECTORY	
DOMAIN	DOMAIN
EDITION	
FUNCTION	FUNCTION
HIERARCHY	HIERARCHY
INDEX	
INDEXTYPE	INDEXTYPE
INMEMORY JOIN GROUP	INMEMORY JOIN GROUP
JAVA	JAVA
LIBRARY	LIBRARY
MATERIALIZED VIEW	MATERIALIZED VIEW
MATERIALIZED VIEW LOG	MATERIALIZED VIEW LOG
MATERIALIZED ZONEMAP	MATERIALIZED ZONEMAP
MLE ENV	MLE ENV
MLE MODULE	MLE MODULE
OPERATOR	OPERATOR
PACKAGE	PACKAGE
PACKAGE BODY	
PROCEDURE	PROCEDURE
PROPERTY GRAPH	PROPERTY GRAPH
SEQUENCE	SEQUENCE
SYNONYM	SYNONYM
TABLE	TABLE
TABLESPACE	TABLESPACE
TRIGGER	TRIGGER
TYPE	TYPE
TYPE BODY	
USER	USER

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-8 (Cont.) Object Types Supported for CREATE, ALTER, and DROP Commands

CREATE and DROP Commands	ALTER Command
VIEW	VIEW

Limitations for IF [NOT] EXISTS

A subset of statements for the supported object types that are listed in Table 8-8 cannot be used with `IF [NOT] EXISTS`. These are DDL statements that may move data, add or drop partitions, revalidate a materialized view, or create or alter a private temporary table. In such cases, a custom error is raised, informing the user that the particular DDL statement cannot be used with `IF [NOT] EXISTS`.

The following are the statements that cannot be used with `IF [NOT] EXISTS`:

- `ALTER TABLE <table> ADD|DROP PARTITION ...`
- `ALTER TABLE <table> COALESCE PARTITION ...`
- `ALTER TABLE <table> MOVE ...`
- `ALTER MATERIALIZED VIEW <mat_view> MERGE PARTITIONS ...`
- `ALTER MATERIALIZED VIEW <mat_view> COMPILE ...`
- `CREATE TEMPORARY TABLE ...`
- `DROP TEMPORARY TABLE ...`

8.10.5 Limitations for `CREATE OR REPLACE` Statements

The `IF NOT EXISTS` clause is not allowed with the `CREATE OR REPLACE` syntax.

Here are some examples that illustrate how the `CREATE OR REPLACE` statement and the `CREATE` statement can or cannot be used with the `IF NOT EXISTS` clause:

```
-- not allowed, REPLACE cannot coexists with IF NOT EXISTS
CREATE OR REPLACE SYNONYM IF NOT EXISTS t1_syn FOR t1;

-- allowed
CREATE SYNONYM IF NOT EXISTS t1_syn FOR t1;

-- allowed
CREATE OR REPLACE SYNONYM t1_syn FOR t1;
```

8.10.6 SQL*Plus Output Messages for DDL Statements

To enable support for writing idempotent DDL scripts, the SQL*Plus output messages for the `CREATE`, `ALTER`, and `DROP` statements with `IF EXISTS` and `IF NOT EXISTS` commands, indicate a successful result even when an internal error related to an object's existence occurs. The suppression of error messages is intended by design, to allow users to write idempotent DDL statements. For instance, the following two statements, when executed one after another, generate the same 'Table created' output message, although the second statement does not create a new table.

```
CREATE TABLE T1 (COL NUMBER);
> Table created.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CREATE TABLE IF NOT EXISTS T1 (COL NUMBER);
> Table created.
```

There is no difference in the two output messages: the first, where the statement actually creates the table, and the second, where the table is not created because the table already exists. The second statement results in a no-op and suppresses the error. To know if an object already exists or not, you can run the DDL statements without the `IF [NOT] EXISTS` clause.

Copyright © 1999, 2025, Oracle and/or its affiliates.

9

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.

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

> **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.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for information about data type conversion
> - PL/SQL Data Types
> - Data Types
> - Overview of Precompilers

9.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- How the Correct Data Type Improves Performance

9.1.1 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**
>
> Maintaining Data Integrity in Database Applications, for information about data integrity and constraints

9.1.2 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 x 4,000 x 100—almost 4 MB. In contrast, if the column length is 80, the RAM that your application must allocate is 10 x 80 x 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. 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database SQL Language Reference* for more information about `ALTERTABLE`
- *Oracle Database SQL Language Reference* for more information about extended data types

9.1.3 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.

Example 9-1 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 9-1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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','yyyymmdd')
  AND     TO_DATE('20010101','yyyymmdd');
  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 |       |
|   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.

See Also

- EXPLAIN PLAN Statement
- *Oracle Database SQL Tuning Guide* for more information about Full Table Scans
- *Oracle Database SQL Tuning Guide* for more information about Index Range Scans

Copyright © 1999, 2025, Oracle and/or its affiliates.

9.2 Representing Character Data

Table 9-1 summarizes the SQL data types that store character data.

Table 9-1 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.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database SQL Language Reference* for more information about comparison semantics for these data types
- Large Objects (LOBs) for more information about `CLOB` and `NCLOB` data types
- LONG and LONG RAW Data Types for more information about `LONG` data type

9.3 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 x 10^{-130} through 9.99 x10^{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).

Note

Oracle recommends using `BINARY_FLOAT` and `BINARY_DOUBLE` instead of `FLOAT`, a subtype of `NUMBER` .

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database SQL Language Reference* for more information about data types

9.3.1 Floating-Point Number Components

The formula for a floating-point value is:

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

For example, the floating-point value 4.31 is represented:

$(-1)^{0}.431.10^{-2}$

The components of the preceding representation are:

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

9.3.2 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 9-2 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* * $10^{exponent}$.)

Table 9-2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 9-2 (Cont.) 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
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*.

9.3.2.1 Binary Floating-Point Formats

This formula determines the value of a floating-point number that uses a binary format:

$(-1)^{sign}\ 2^{E}\ (bit_0\ bit_1\ bit_2\ \ldots\ bit_{p-1})$

Table 9-3 describes the components of the preceding formula.

Table 9-3 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 9-4 shows.

Table 9-4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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*.

9.3.3 Representing Special Values with Native Floating-Point Data Types

The IEEE 754 standard supports the special values shown in Table 9-5.

Table 9-5 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 9-5 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 `NaN`s (which do not raise additional exceptions as they propagate through most operations) and signaling `NaN`s (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 `NaN`s. 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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).

9.3.4 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.

Comparisons ignore the sign of zero (-0 equals +0).

> **See Also**
>
> - *Oracle Database SQL Language Reference* for more information about Numeric Precedence
> - *Oracle Database SQL Language Reference* for more information about Comparison Conditions

9.3.5 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database SQL Language Reference* for general information about arithmetic operations

9.3.6 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 9-6.

Table 9-6 Values Resulting from Exceptions

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

9.3.7 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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

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*.

9.4 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 9-7 summarizes the SQL datetime data types.

Table 9-7 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.
`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Call Interface Programmer's Guide* for more information about Oracle Database internal date types
> - *Oracle Database SQL Language Reference* for more information about date and time data types

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

9.4.1 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 9-2 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 9-2 Displaying Current Date and Time

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

Result:

```
NOW
----------------------
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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`.

> **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
> - *Oracle Database Reference*for more information about `FIXED_DATE`

9.4.2 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 9-3 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 9-3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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

9.4.3 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 9-4 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,

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 9-4 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.
```

9.4.4 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. .

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database SQL Language Reference* for the complete list of datetime functions
- *Oracle Database SQL Language Reference*

9.4.5 Conversion Functions for Datetime Data Types

Table 9-8 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 9-8 SQL Conversion Functions for Datetime Data Types

Function	Converts ...	To ...
NUMTODSINTERVAL	NUMBER	INTERVAL DAY TO SECOND
NUMTOYMINTERVAL	NUMBER	INTERVAL DAY TO MONTH
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

Copyright © 1999, 2025, Oracle and/or its affiliates.

9.4.6 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.

9.5 Representing Specialized Data

Topics:

- Representing Spatial Data
- Representing Large Amounts of Data
- Representing Searchable Text
- Representing XML Data
- Representing Dynamically Typed Data
- Representing ANSI_ DB2_ and SQL/DS Data

9.5.1 Representing Spatial Data

Spatial data is used by location-enabled applications, geographic information system (GIS) applications, and geoimaging applications.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about representing spatial data in Oracle Database

9.5.2 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)

9.5.2.1 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 9-9 summarizes the LOBs.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 9-9 Large Objects (LOBs)

Data Type	Description
`BLOB`	**Binary large object** Stores any kind of data in binary format. Typically used for multimedia data such as images, audio, and video.
`CLOB`	**Character large object** Stores string data in the database character set format. Used for large strings or documents that use the database character set exclusively.
`NCLOB`	**National character large object** Stores string data in National Character Set format. Used for large strings or documents in the National Character Set.
`BFILE`	**External large object** Stores a binary file outside the database in the host operating system file system. Applications have read-only access to `BFILE`s. Used for static data that applications do not manipulate, such as image data. 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.

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
- *Oracle Database SQL Language Reference* for more information about LOB functions

9.5.2.2 LONG and LONG RAW Data Types

Note

All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

For more information, see:

Migrating Columns from LONGs to LOBs

`LONG` columns store variable-length character strings containing up to 2 gigabytes - 1 bytes. .

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **See Also**
>
> *Oracle Database SQL Language Reference* for more information about data types

9.5.3 Representing JSON Data

A new JSON data type to natively store JSON data in Oracle Database.

Oracle Database stores JSON data type to natively store JSON data.

If you have information stored in JSON format, then you can use the Oracle-supplied type `JSON` type to natively store in the database. Oracle provides indexing, a rich set of packages and operators that can operate on JSON data.

With `JSON` values, you can use:

- `JSON` type standard functions
- `IS_JSON/IS_NOT_JSON` constraints

> **See Also**
>
> - Oracle Database JSON Developer's Guide for information about Oracle JSON data type and how you can use it to store, generate, manipulate, manage, and query JSON data in the database

9.5.4 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

9.5.5 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`.

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*)

Copyright © 1999, 2025, Oracle and/or its affiliates.

- PL/SQL `DBMS_XML` packages (see *Oracle AI 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*
> - *Oracle Database SQL Language Reference* for more information about `XMLType`

9.5.6 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.

To check and manipulate type information, use the `DBMS_TYPES` package, as in Example 9-5.

With OCI, use the `OCIAnyData` and `OCIAnyDataSet` interfaces.

Example 9-5 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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.TYPECODE_NUMBER 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.TYPECODE_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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database Object-Relational Developer's Guide* for more information about these ADTs
- *Oracle Call Interface Programmer's Guide* for more information about `OCIAnyData` and `OCIAnyDataSet` interfaces
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_TYPES` package

9.5.7 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.

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.

See Also

Oracle Database SQL Language Reference for conversion details

9.6 Identifying Rows by Address

The fastest way to access the row of a database table is by its address, or **rowid**. 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`.

Note

The rowid for a row may change for a number of reasons, which may be user initiated or internally by the database engine. You cannot depend on the rowid to be pointing to the same row or a valid row at all after any of these operations has occurred.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
- *Oracle Database Concepts* for more information about HCC

9.7 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

9.7.1 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, or ARG*n*. ARG*n* is the data type of the *n*th argument of the function. For example:

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

See Also

- SQL Data Type Families
- *Oracle Database SQL Language Reference* for more information about MAX function
- *Oracle Database SQL Language Reference* for more information about DECODE function

9.7.2 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 9-10 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

9.7.3 SQL Data Type Families

Often, a SQL function argument can have any data type in a data type family. Table 9-11 shows the SQL data type families and their member data types.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 9-11 SQL Data Type Families

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

10

Registering Application Data Usage with the Database

This chapter details how you can use centralized database-centric entities called data use case domains and schema annotations to register information about the intended application data usage.

Oracle AI Database 26ai introduces a centralized, database-centric approach to handling intended data usage information using data use case domains and schema annotations. You can add data use case domains and schema annotations centrally in the database to register data usage intent, which is then accessible to various applications and tools.

See Also

- *Oracle Database Concepts* for information about Data Use Case Domains and Schema Annotations.
- *Oracle Database SQL Language Reference* for the syntactic and semantic information about data use case domains and schema annotations.

Sections:

- Data Use Case Domains
- Schema Annotations

10.1 Data Use Case Domains

This section explains how you can use data use case domains (hereinafter "use case domains") in your applications.

Topics:

- Overview of Use Case Domains
- Use Case Domain Types and When to Use Them
- Privileges Required for Use Case Domains
- Using a Single-column Use Case Domain
- Using a Multi-column Use Case Domain
- Using a Flexible Use Case Domain
- Using an Enumeration Use Case Domain
- Specifying a Data Type for a Domain
- Changing the Use Case Domain Properties
- SQL Functions for Use Case Domains
- Viewing Domain Information

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Built-in Use Case Domains

10.1.1 Overview of Use Case Domains

Use case domains are lightweight usage specifiers with the optional database-side enforcement that applications can use to centrally document intended data usage. As a high-level dictionary object, a use case domain includes built-in usage properties associated with table columns, such as the default value, check constraints, collations, display and order formats, and annotations. Using centralized domain information, applications can standardize operations without depending on application-level metadata. For example, you can use use case domains to mask credit card numbers or to format phone numbers and currency values.

As a database object, a use case domain belongs to a schema and provides a common column definition without modifying the underlying column data type. A use case domain encapsulates some common characteristics of a table column into a reusable object that can be reused on other table columns without having to repeat these characteristics.

For example, a schema may have many tables with columns that hold email addresses, such as billing emails, invoice emails, and customer contact emails. Email addresses have a special format because they require the "@" sign. You can define a use case domain for the email address using a check constraint such as `regexp_like (email_dom, '^(\S+)\@(\S+)\.(\S+)$')`, and associate the domain with other email columns to have your application display the associated column's email addresses with the standard "@" sign prior to the domain name. Likewise, you may want to standardize display formats for vehicle license plates, which may require hyphens to separate the number from the other information. Using a use case domain, you can show the license information as "ABC-123" even when the license column has a varchar2(6) data type.

Benefits of using a use case domain include: improved data quality because it enables you to handle values consistently across applications; reduced coding because you can reuse a column's properties and constraints across your application; and consistency and clarity in operations because applications can inquire about these use case domains to understand what data they are operating against.

> **See Also**
>
> *Oracle Database Concepts* for more information about Use Case Domains.

10.1.2 Use Case Domain Types and When to Use Them

There are three different types of domains: single-column, multi-column, and flexible domains.

Single-column Use Case Domains

A single-column use case domain is created for one column, and is used when you want to make a column usage definition consistent throughout your application. For example, you can create a single-column domain for email addresses, postal codes, or vehicle numbers.

Unless prefixed with multi-column or flexible, a "use case domain" or "domain" means a single-column use case domain for the purposes of this document.

Multi-column Use Case Domains

A multi-column use case domain creates column usage definitions for multiple columns under a single domain. A multi-column use case domain is ideal when you have logical entities

Copyright © 1999, 2025, Oracle and/or its affiliates.

spanning across two or more table columns. For example, you can use a multi-column use case domain to represent an address.

Flexible Use Case Domains

A flexible use case domain is a domain that dynamically selects from a set of single-column or multi-column use case domains based on a discriminant column's values. A flexible domain assigns a specific domain (also known as a constituent domain) to a set of value columns based on a mapping expression composed from one or more discriminant columns.

In addition to these use case domain types, you can also use built-in use case domains directly on table columns. Some examples are `email_d`, `ssn_d`, and `credit_card_number_d`.

See Also

- *Oracle Database Concepts* for more information about types of Data Use Case Domains.
- Built-in Use Case Domains

10.1.3 Privileges Required for Use Case Domains

To work with use case domains, you require the following privileges:

Note

The Database Administrator (DBA) role includes all the following privileges.

DDL Privilege	Action
CREATE DOMAIN	You can create a domain in your own schema. The RESOURCE and DB_DEVELOPER_ROLE roles include the CREATE DOMAIN privilege.
CREATE ANY DOMAIN	You can create a domain in any schema.
ALTER ANY DOMAIN	You can alter a domain in any schema.
DROP ANY DOMAIN	You can drop a domain in any schema.
EXECUTE ANY DOMAIN	You can reference or use a domain in any schema. To explicitly grant execute privileges to a user on a domain in any schema, use the following code: `GRANT EXECUTE ON <schemaName.domainName> TO <user>;`

10.1.4 Using a Single-column Use Case Domain

This section explains how you can create, associate, alter, disassociate, and drop a single-column use case domain (hereinafter "use case domain" or "domain").

Copyright © 1999, 2025, Oracle and/or its affiliates.

Topics

- Creating a Use Case Domain
- Associating Use Case Domains with Columns at Table Creation
- Associating Use Case Domains with Existing or New Columns
- Altering a Use Case Domain
- Disassociating a Use Case Domain from a Column
- Dropping a Use Case Domain

10.1.4.1 Creating a Use Case Domain

You can define a use case domain that encapsulates a set of optional properties and constraints to represent common values, such as email addresses and credit card numbers.

The following are some examples of creating use case domains, where a domain is based on the properties of a table column, such as default value, constraints, annotations, or display and order expressions.

Example 10-1 Creating a Use Case Domain for Hourly Wages

A Human Resource Application (HRA) creates many tables for different companies. Most companies have a column that stores hourly wages. The HRA can create a use case domain for the hourly wages columns as follows:

```
CREATE DOMAIN HourlyWages AS NUMBER
  DEFAULT ON NULL 15
  CONSTRAINT MinimalWage CHECK (HourlyWages > = 7 AND HourlyWages <=1000)
ENABLE
  DISPLAY TO_CHAR(HourlyWages, '$999.99')  ORDER ( -1*HourlyWages )
  ANNOTATIONS (properties '{"Purpose": "Wages", "Applicability": "USA",
"Industry": {"Sales", "Manufacturing"} }');
```

Example 10-2 Creating a Use Case Domain for Surrogate Keys

If you want to annotate surrogate key columns in your database application and maintain standard meta-data for such columns, create a domain similar to the following:

```
CREATE DOMAIN surrogate_id AS INTEGER
  STRICT
  NOT NULL
  ANNOTATIONS ( primary_key, mandatory, operations '["insert", "delete"]' );
```

Example 10-3 Creating a Use Case Domain for Birth Dates

The following is a use case domain that ensures that all columns with birth dates are "date only" and displays the age in years.

```
CREATE DOMAIN birth_date AS DATE
  CONSTRAINT birth_date_only_c check ( birth_date = trunc ( birth_date ) )
  DISPLAY FLOOR ( months_between ( sysdate, birth_date ) / 12 ) || ' years'
  ANNOTATIONS ( sensitive 'PII Data', operations '["insert", "update"]' );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-4 Creating a Use Case Domain for Default Date and Time Values

You can create a use case domain for date and time values to ensure that the inserts are in a standard format.

```
CREATE DOMAIN insert_timestamp AS
  TIMESTAMP WITH LOCAL TIME ZONE
  DEFAULT ON NULL systimestamp;
```

Example 10-5 Creating a Use Case Domain for Positive Heights

The following use case domain ensures that people have positive heights and the heights are sorted in a descending order.

```
CREATE DOMAIN height AS NUMBER
  CONSTRAINT positive_height_c CHECK ( value > 0 )
  ORDER value * -1
  ANNOTATIONS ( operations '["insert", "update"]' );
```

Example 10-6 Creating a Use Case Domain for Positive Weights

The following use case domain ensures that people have positive weights.

```
CREATE DOMAIN weight AS NUMBER
  CONSTRAINT positive_weight_c CHECK ( value > 0 )
  ANNOTATIONS ( operations '["insert", "update"]' );
```

Example 10-7 Defining a Domain with Multiple Check Constraints

You can define a use case domain for email addresses, similar to the one in the following example, which also has multiple `CHECK` constraints.

```
CREATE SEQUENCE IF NOT EXISTS email_seq;

CREATE DOMAIN email AS VARCHAR2(100)
  DEFAULT ON NULL email_seq.NEXTVAL || '@domain.com'
  CONSTRAINT email_c CHECK (REGEXP_LIKE (email, '^(\S+)\@(\S+)\.(\S+)$'))
  CONSTRAINT email_max_len_c CHECK (LENGTH(email) <=100) DEFERRABLE INITIALLY
DEFERRED
  DISPLAY '---' || SUBSTR(email, INSTR(email, '@') + 1);
```

Any column of `VARCHAR2(L [BYTE|CHAR])` data type that satisfies both constraints can be associated with the domain. The `INITIALLY DEFERRED` clause delays validation of the constraint `email_max_len_c` values until commit time.

Example 10-8 JSON Schema Validation

A domain can also be used for reusable JSON schema validation, as in the following example.

```
CREATE DOMAIN department_json_doc AS JSON
  CONSTRAINT CHECK (
    department_json_doc IS JSON VALIDATE USING '{
      "type": "object",
      "properties": {
        "departmentName": { "type": "string" },
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      "employees": { "type": "array" }
    },
    "required" : [ "departmentName", "employees" ],
    "additionalProperties": false
  }' );
```

See Also

- *Oracle Database SQL Language Reference* for the syntactic and semantic information on creating a use case domain: CREATE DOMAIN
- Specifying a Data Type for a Domain

10.1.4.2 Associating Use Case Domains with Columns at Table Creation

After defining a use case domain, you can associate the domain with new table columns when creating a table or associate it with existing columns in an existing table. Associating a use case domain with a column explicitly applies the domain's optional properties and constraints to the column.

You can use the `CREATE TABLE` DDL to associate a use case domain with a newly created column. The following examples show how you can associate a use case domain with columns in a new table. These examples use the use case domains created in the earlier examples.

Example 10-9 Associating `HourlyWages` Use Case Domain at Table Creation

Using the `HourlyWages` domain, the HRA can create multiple tables where wage columns have the same domain characteristics.

```
CREATE TABLE employee (
  name VARCHAR2(100),
  id NUMBER,
  wage NUMBER DOMAIN HourlyWages);


CREATE TABLE wage (
  name VARCHAR2(100),
  id NUMBER,
  wage NUMBER DOMAIN HourlyWages,
  gross_pay NUMBER,
  deductions NUMBER,
  net_pay NUMBER);
```

Example 10-10 Associating the `surrogate_id` Use Case Domain at Table Creation

When you associate a strict use case domain such as `surrogate_id` with columns, ensure that the associated columns have the same data type as the domain. A strict domain also requires that the column length, scale, and precision match with the domain.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The following code fails with `ORA-11517: the column data type does not match the domain column` because you are trying to link a `NUMBER` data type column with a `INTEGER/NUMBER(*,0)` data type domain.

```
CREATE TABLE orders (
  order_id NUMBER DOMAIN surrogate_id,
  customer_id NUMBER,
  order_datetime TIMESTAMP WITH LOCAL TIME ZONE
    DEFAULT SYSTIMESTAMP);
```

To ensure that the association works, you can use the `NUMBER(*,0)` column data type to associate with the `surrogate_id` use case domain (`INTEGER == NUMBER(*,0)`).

```
CREATE TABLE orders (
  order_id NUMBER(*,0) DOMAIN surrogate_id,
  customer_id NUMBER,
  order_datetime TIMESTAMP WITH LOCAL TIME ZONE
    DEFAULT SYSTIMESTAMP);
```

Example 10-11 Associating the `surrogate_id`, `birth_date`, `height`, and `weight` Use Case Domains at Table Creation

The `DOMAIN` keyword is optional. You can see in the following example that the `birth_date` domain is associated with `date_of_birth` column without the `DOMAIN` keyword. The example also shows that you can define a more precise data type for the columns with regards to the precision and scale than what is in the domain. The `height_in_cm` and `weight_in_kg` columns have their associated domain data type as `NUMBER` whereas the column data type has precision and scale values, such as `NUMBER(4,1)`.

```
CREATE TABLE people (
  person_id     DOMAIN surrogate_id
    GENERATED BY DEFAULT AS IDENTITY
    PRIMARY KEY,
  full_name     VARCHAR2(256),
  date_of_birth birth_date,
  height_in_cm  NUMBER(4, 1) DOMAIN height,
  weight_in_kg  NUMBER(6, 3) DOMAIN weight);
```

Example 10-12 Associating the `department_json_doc` Use Case Domain at Table Creation

```
CREATE TABLE departments (
  department_id  INTEGER PRIMARY KEY,
  department_doc JSON DOMAIN department_json_doc);
```

Guidelines

- When associating a domain with a column, you can specify the domain name in addition to the column's data type, in which case the column's data type is used, provided that the domain data type is compatible with the column's data type.
- When associating a domain with a column, you can specify a domain name instead of the column's data type, in which case the domain data type is used for the column, wherein the `DOMAIN` keyword is optional.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- If a domain is defined as `STRICT`, the domain's data type, scale, and precision must match the column's data type, scale, and precision.
- If a domain is not defined as `STRICT`, you can associate a domain of any length with a column of any length. For instance, you can associate a domain of `VARCHAR2(10)` with any `VARCHAR2` column.

See Also

- *Oracle Database SQL Language Reference* for the syntactic and semantic information about creating a use case domain: CREATE DOMAIN
- Specifying a Data Type for a Domain for information about column and domain data types

10.1.4.2.1 Using DML on Columns Associated with Use Case Domains

The following are some examples of DML statements that you can use on the newly created table columns with associated use case domains.

Example 10-13 Using DML Statements on the `people` table

The following `INSERT` commands insert data into the `people` table columns, while verifying that the check constraint specified in the associated domain is not violated.

```
INSERT INTO people
  VALUES ( 1, 'Sally Squirell', date'1981-01-01', 180.1, 61 );

INSERT INTO people
  VALUES ( 2, 'Brian Badger', date'2016-12-31', 120.4, 27.181 );
```

The following `INSERT` command fails because `height` is specified as a negative number, and hence the associated check constraint is violated.

```
INSERT INTO people
  VALUES ( 3, 'Fergal Fox', date'2023-04-12', -99, 1 );
```

The output is:

```
ORA-11534: check constraint (HR.SYS_C009232) due to domain constraint
HR.POSITIVE_HEIGHT_C of domain HR.HEIGHT violated
```

You can use the associated domains to display data with the heights sorted in a descending order and also view the corresponding age and weight.

```
SELECT full_name, DOMAIN_DISPLAY ( date_of_birth ) age,
       height_in_cm, weight_in_kg
  FROM people
  ORDER BY DOMAIN_ORDER ( height_in_cm );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> SQL Functions for Use Case Domains for details about domain functions, such as `DOMAIN_DISPLAY` and `DOMAIN_ORDER`.

The output is:

```
FULL_NAME            AGE        HEIGHT_IN_CM WEIGHT_IN_KG
-------------------- ---------- ------------ ------------
Sally Squirell       42 years          180.1           61
Brian Badger         6 years           120.4       27.181
```

You can describe the `people` table to see the data type definition and the referenced domain information.

```
DESC people;
```

The output is:

```
 Name                  Null?    Type
 --------------------- -------- ----------------------------------------
 PERSON_ID             NOT NULL NUMBER(38) HR.SURROGATE_ID
 FULL_NAME                      VARCHAR2(256)
 DATE_OF_BIRTH                  DATE HR.BIRTH_DATE
 HEIGHT_IN_CM                   NUMBER(4,1) HR.HEIGHT
 WEIGHT_IN_KG                   NUMBER(6,3) HR.WEIGHT
```

The following `SELECT` command enables you to view the column annotations that are inherited from the associated domains.

```
SELECT column_name, annotation_name, annotation_value
  FROM user_annotations_usage
  WHERE object_name = 'PEOPLE';
```

The output is:

```
COLUMN_NAME          ANNOTATION_NAME      ANNOTATION_VALUE
-------------------- -------------------- --------------------
PERSON_ID            PRIMARY_KEY          <null>
PERSON_ID            MANDATORY            <null>
PERSON_ID            OPERATIONS           ["insert", "delete"]
DATE_OF_BIRTH        OPERATIONS           ["insert", "update"]
DATE_OF_BIRTH        SENSITIVE            PII Data
HEIGHT_IN_CM         OPERATIONS           ["insert", "update"]
WEIGHT_IN_KG         OPERATIONS           ["insert", "update"]
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-14 Using DML Statements on the `departments` table with JSON data

The following `INSERT` command on the `departments` table succeeds because it includes all the JSON attributes in the `department_json_doc` domain.

```
INSERT INTO departments
  VALUES ( 1, '{
    "departmentName" : "Accounting",
    "employees" : [
      {"empName":"William"},
      {"empName":"Shelley"}
    ]
  }');
```

The following `INSERT` command fails with `ORA-40875: JSON schema validation error - missing employees attribute` because the `employees` attribute is missing.

```
INSERT INTO departments
  VALUES ( 2, '{
    "departmentName" : "Finance"
  }');
```

The following `INSERT` command fails with `ORA-40875: JSON schema validation error - extra manager attribute` because the `manager` attribute is not found in the associated `department_json_doc` use case domain.

```
INSERT INTO departments
  VALUES ( 3, '{
    "departmentName" : "Executive",
    "employees" : [
      {"empName":"Lex"},
      {"empName":"Neena"}
    ],
    "manager" : {"empName":"Lex"}
  }');
```

10.1.4.3 Associating Use Case Domains with Existing or New Columns

You can use the `ALTER TABLE` DDL with the `MODIFY` or `ADD` clause to associate a use case domain with an existing or a newly added column.

Example 10-15 Associating a Use Case Domain with a Newly Created Column

For a newly created `customers` table:

```
CREATE TABLE customers (
  cust_id NUMBER,
  cust_email VARCHAR2(100));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

You can use ADD to add a new column: cust_new_email and associate it with the email domain.

```
ALTER TABLE customers
  ADD (cust_new_email VARCHAR2(100) DOMAIN email);
```

Example 10-16 Associating the email Use Case Domain with an Existing Column

You can use MODIFY to modify an existing column: cust_email and associate it with the email domain.

```
ALTER TABLE customers
  MODIFY (cust_email  VARCHAR2(100) DOMAIN email);
```

You can also add a use case domain to a column using the ALTER TABLE ... MODIFY statement. In the following example, the orders table column, namely order_datetime and the insert_timestamp domain have different defaults. The insert_timestamp domain has the DEFAULT with ON NULL clause, which is missing in the order_datetime column. Therefore, when you try to associate the domain with the column, you get an error ORA-11501: The column default does not match the domain default of the column.

```
ALTER TABLE orders
  MODIFY order_datetime DOMAIN insert_timestamp;
```

To overcome the default mismatch, specify a column default clause that matches the domain default.

```
ALTER TABLE orders
  MODIFY order_datetime DOMAIN insert_timestamp
    DEFAULT ON NULL systimestamp;
```

Example 10-17 Querying to View Associated Domains

```
SELECT constraint_name, search_condition_vc, domain_constraint_name
  FROM  user_constraints
  JOIN  user_cons_columns
  USING ( constraint_name, table_name )
  WHERE  table_name = 'PEOPLE'
    AND  constraint_type = 'C'
    AND  column_name = 'WEIGHT_IN_KG';
```

The output is:

```
CONSTRAINT_NAME      SEARCH_CONDITION_VC            DOMAIN_CONSTRAINT_NAME
-------------------- ------------------------------ ------------------------
SYS_C008493          "WEIGHT_IN_KG">0               POSITIVE_WEIGHT_C
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Guidelines

- The `DOMAIN` keyword is optional for the `ALTER TABLE .. ADD` statement if a domain only is specified for the newly added column.
- The `DOMAIN` keyword is mandatory for the `ALTER TABLE .. MODIFY` statement.
- The column data type should be compatible with the domain data type.
- If a domain has default expression or collation, it should match the associated column's default expression and collation.
- If the associated column already has a domain associated with it, an error is returned.

10.1.4.4 Altering a Use Case Domain

You can alter a use case domain for its display expression, order expression, and annotation. The following `ALTER DOMAIN` DDL statements are supported.

Example 10-18 Removing the Order Expression from a Domain

The `height` use case domain definition has the order expression as `VALUE`, so to remove the order expression, you must drop the order expression.

```
ALTER DOMAIN height DROP ORDER;
```

Example 10-19 Adding a Display Expression to a Use Case Domain

To add a display expression to the `height` use case domain, use the following `ALTER` command.

```
ALTER DOMAIN height
  ADD DISPLAY round ( value ) || ' cm';
```

Example 10-20 Changing the Display Expression of the `birth_date` Domain

To change the display expression of the `birth_date` domain to years and months, use the following `ALTER` command.

```
ALTER DOMAIN birth_date
  MODIFY DISPLAY
    FLOOR ( months_between ( sysdate, birth_date ) / 12 ) || ' years ' ||
    MOD ( FLOOR ( months_between ( sysdate, birth_date ) ), 12 ) || ' months';
```

Example 10-21 Querying the `people` table for the Altered `birth_date`, `height`, and `weight` Domains

```
COLUMN age FORMAT A20

SELECT full_name, DOMAIN_DISPLAY ( date_of_birth ) age,
       DOMAIN_DISPLAY ( height_in_cm ) height_in_cm, weight_in_kg
  FROM  people
  ORDER  BY DOMAIN_ORDER ( height_in_cm );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> SQL Functions for Use Case Domains for details about domain functions, such as DOMAIN_DISPLAY and DOMAIN_ORDER.

The output is:

```
FULL_NAME            AGE                  HEIGHT_IN_CM  WEIGHT_IN_KG
-------------------- -------------------- ------------ ------------
Sally Squirell       42 years 5 months    180 cm                 61
Brian Badger         6 years 5 months     120 cm              27.181
```

Example 10-22 Changing the Annotations on a Use Case Domain

The following code adds the annotations for the height use case domain.

```
ALTER DOMAIN height
  ANNOTATIONS (
    operations '["insert", "update", "sort"]',
    sensitive 'Private data');
```

Example 10-23 Querying the Dictionary Views for the Annotation Changes

```
COLUMN annotation_value FORMAT A40

SELECT column_name, annotation_name, annotation_value
  FROM user_annotations_usage
  WHERE object_name = 'PEOPLE';
```

The output is:

```
COLUMN_NAME          ANNOTATION_NAME
ANNOTATION_VALUE
-------------------- --------------------
----------------------------------------
PERSON_ID            PRIMARY_KEY
<null>
PERSON_ID            MANDATORY
<null>
PERSON_ID            OPERATIONS          ["insert",
"delete"]
DATE_OF_BIRTH        SENSITIVE           PII
Data
DATE_OF_BIRTH        OPERATIONS          ["insert",
"update"]
HEIGHT_IN_CM         OPERATIONS          ["insert",
"update"]
HEIGHT_IN_CM         OPERATIONS          ["insert", "update",
"sort"]
HEIGHT_IN_CM         SENSITIVE           Private
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
data
WEIGHT_IN_KG         OPERATIONS           ["insert",
"update"]
```

Guidelines

- You can alter the display expression of a domain only if the domain is not constituent in a flexible domain.
- You can alter the order expression of a domain only if the domain is not constituent in a flexible domain.
- You can alter only domain-level annotations.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for the syntactic and semantic information about altering a use case domain: ALTER DOMAIN.
> - Viewing Domain Information for the use case domain dictionary views

10.1.4.5 Disassociating a Use Case Domain from a Column

You can use the `ALTER TABLE` DDL with the `DROP` clause to disassociate a use case domain from a column.

Example 10-24 Disassociating a Use Case Domain from a Column

To drop the associated domain from the `cust_email` column of the `customers` table:

```
ALTER TABLE customers
  MODIFY ( cust_email ) DROP DOMAIN;
```

To drop the domain but keep the domain's constraint:

```
ALTER TABLE customers
  MODIFY ( cust_email ) DROP DOMAIN PRESERVE CONSTRAINTS;
```

Example 10-25 Disassociating a Use Case Domain from a Column

The following code removes the domain association of the `height` domain from the `height_in_cm` column in the `people` table, while preserving the constraint.

```
ALTER TABLE people
  MODIFY ( height_in_cm ) DROP DOMAIN PRESERVE CONSTRAINTS;
```

Example 10-26 Querying the Columns with Disassociated Use Case Domain

The `SELECT` query on the `user_constraints` dictionary table reveals the changes.

```
SELECT constraint_name, search_condition_vc, domain_constraint_name
  FROM user_constraints
  JOIN user_cons_columns
  USING ( constraint_name, table_name )
  WHERE table_name = 'PEOPLE'
    AND constraint_type = 'C';
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The output is:

```
CONSTRAINT_NAME      SEARCH_CONDITION_VC
DOMAIN_CONSTRAINT_NAME
-------------------- ----------------------------------------
------------------------------
SYS_C009491          "DATE_OF_BIRTH"=TRUNC("DATE_OF_BIRTH")
BIRTH_DATE_ONLY_C
SYS_C009494          "WEIGHT_IN_KG">0
POSITIVE_WEIGHT_C
SYS_C009489          "PERSON_ID" IS NOT NULL
<null>
SYS_C009490          "HEIGHT_IN_CM">0
<null>
```

Example 10-27 Re-adding the Removed `height` Domain

The following code re-adds the removed `height` domain to the `height_in_cm` column in the `people` table.

```
ALTER TABLE people
  MODIFY ( height_in_cm ) ADD DOMAIN height;
```

Example 10-28 Querying the Re-added Domain

A duplicate `height_in_cm > 0` constraint is created with the re-added `height` domain.

```
SELECT constraint_name, search_condition_vc, domain_constraint_name
  FROM user_constraints
  JOIN user_cons_columns
  USING ( constraint_name, table_name )
  WHERE table_name = 'PEOPLE'
    AND constraint_type = 'C'
  ORDER BY search_condition_vc;
```

The output is:

```
CONSTRAINT_NAME      SEARCH_CONDITION_VC
DOMAIN_CONSTRAINT_NAME
-------------------- ----------------------------------------
------------------------------
SYS_C009491          "DATE_OF_BIRTH"=TRUNC("DATE_OF_BIRTH")
BIRTH_DATE_ONLY_C
SYS_C009495          "HEIGHT_IN_CM">0
POSITIVE_HEIGHT_C
SYS_C009490          "HEIGHT_IN_CM">0
<null>
SYS_C009489          "PERSON_ID" IS NOT NULL
<null>
SYS_C009494          "WEIGHT_IN_KG">0
POSITIVE_WEIGHT_C
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Guidelines

On dropping the domain for a column, the following are preserved by default:

- The domain's collation.
- The non-domain constraint that is added to the column.

The domain's default value is not preserved. It is only kept if the default is explicitly applied to the column.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for the syntactic and semantic information about altering a use case domain: ALTER DOMAIN.
> - Viewing Domain Information for the use case domain dictionary views

10.1.4.6 Dropping a Use Case Domain

You can drop a use case domain. See *Oracle Database SQL Language Reference* for more information about dropping a domain.

Example 10-29 Dropping a Use Case Domain

The following `customers` table has one of its column associated with the `email` domain:

```
CREATE TABLE customers (
  cust_id NUMBER,
  cust_email VARCHAR2(100) DOMAIN email);
```

The following `DROP` command returns an error because the `customers` table has the `cust_email` column associated with `email` domain.

```
DROP DOMAIN email;
```

The following `DROP` command succeeds:

```
DROP DOMAIN email FORCE;
```

The `cust_email` column is disassociated from the `email` domain and all statements mentioning the `email` domain are invalidated.

Example 10-30 Dropping a Domain but Preserving the Constraint

The following `DROP` command succeeds but preserves the default and constraint expressions.

```
DROP DOMAIN email FORCE PRESERVE;
```

The `cust_email` column retains the default `ON NULL email_seq.NEXTVAL || '@domain.com'` and preserves the constraint after replacing the domain name in the constraint with the column name: `CONSTRAINT email_c CHECK (REGEXP_LIKE (cust_email, '^(\S+)\@(\S+)\.(\S+)$'))`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Guidelines

- To drop a domain that is referenced in a flexible domain, use `DROP DOMAIN` with the `FORCE` option.
- If the domain is not associated with any table column and the domain is not constituent in a flexible domain, the domain is dropped. If the use case domain is in use, the `DROP` statement fails.
- If the domain is associated with any table column, you must use the `FORCE` option to drop the domain. Using the `FORCE` option also:
  - Removes the default expression, if only domain default is set.
  - Preserves the column default expression, if both domain and column defaults are set.
  - Removes the domain annotation from all associated columns.
  - Preserves the collation of any domain associated columns.
  - Invalidates all SQL dependent statements in the cursor cache.
  - Preserves the constraints on any domain associated columns, if `FORCE PRESERVE` is used.

See Also

- *Oracle Database SQL Language Reference* for the syntactic and semantic information about dropping a use case domain: DROP DOMAIN.
- About Drop Domain and Recycle Bin

10.1.4.6.1 About Drop Domain and Recycle Bin

The dropped tables are placed in the recycle bin. To drop a domain associated with tables in the recycle bin, you must use the `FORCE` option with the `DROP` command.

You can restore the dropped tables that are in the recycle bin to the before-the-drop position using the `FLASHBACK TABLE` command. If a table is restored to the before-the-drop position (using `FLASHBACK TABLE TO BEFORE DROP`) after the associated domain has been dropped, then the table has the same default, collation, nullability, and constraints as was there before the drop, except that none of these attributes would be marked as being inherited from the domain.

Example 10-31 Dropping a Use Case Domain Associated with a Dropped Table

The following `DROP` command tries to remove the `weight` domain that is associated with the `weight_in_kg` column of the `people` table.

```
DROP DOMAIN weight;
```

The command returns the following error output:

```
ORA-11502: The domain WEIGHT to be dropped has dependent objects.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

If you drop the `people` table, and then drop the weight domain, it returns an error because the table is still in the recycle bin.

```
DROP TABLE people;
DROP DOMAIN weight;
```

The command returns the following error output:

```
ORA-11502: The domain WEIGHT to be dropped has dependent objects.
```

Removing the `people` table from the recycle bin permanently, and then running the `DROP` command on the `weight` domain, drops the `weight` domain.

```
PURGE TABLE people;
DROP DOMAIN weight;
```

Guidelines

Here are some points to note when dropping domains that are associated with dropped tables (tables in recycle bin):

- While a table is in the recycle bin, the `ALTER` command on the table is not allowed.
- If a table with a domain association is in the recycle bin, the associated domain cannot be dropped and the `DROP DOMAIN` command fails.
- When the `DROP DOMAIN FORCE` and `DROP DOMAIN FORCE PRESERVE` commands are used, the tables in the recycle bin are disassociated from the domain. The database uses the `FORCE PRESERVE` semantics for tables in the recycle bin, even if you only specify `FORCE`.
- If you want to drop the domain that is associated with a table in the recycle bin, you can use the `PURGE TABLE` command to remove a table from the recycle bin and run the `DROP DOMAIN` command to drop the domain.

10.1.5 Using a Multi-column Use Case Domain

This section explains how you can create, associate, alter, disassociate, and drop a multi-column use case domain.

Topics

- Creating a Multi-column Use Case Domain
- Associating a Multi-column Use Case Domain at Table Creation
- Associating a Multi-column Use Case Domain with Existing Columns
- Altering a Multi-column Use Case Domain
- Disassociating a Multi-column Use Case Domain from a Column
- Dropping a Multi-column Use Case Domain

10.1.5.1 Creating a Multi-column Use Case Domain

You can use a multi-column use case domain to group logical entities that span across table columns, such as addresses.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-32 Creating a Multi-column Use Case Domain for Addresses

You can create a multi-column use case domain called "US_city" with three columns for address entries, as follows:

```
CREATE DOMAIN US_city AS (
    name  AS VARCHAR2(30) ANNOTATIONS (Address),
    state AS VARCHAR2(2) ANNOTATIONS (Address),
    zip AS NUMBER ANNOTATIONS (Address)
  )
  CONSTRAINT City_CK CHECK(state in ('CA','AZ','TX') and zip < 100000)
  DISPLAY name||', '|| state ||', '||TO_CHAR(zip)
  ORDER state||', '||TO_CHAR(zip)||', '||name
  ANNOTATIONS (Title 'Domain Annotation');
```

Example 10-33 Creating a Multi-column Use Case Domain for Currency

The following code creates a multi-column use case domain called `currency` that displays monetary values as an amount in a given currency, and the currency codes. The display is sorted by value (low to high), and then by the currency code.

```
CREATE DOMAIN currency AS (
    amount  AS NUMBER,
    iso_currency_code  AS CHAR(3 CHAR)
  )
  DISPLAY iso_currency_code || TO_CHAR ( amount, '999,999,990.00' )
  ORDER TO_CHAR ( amount, '999,999,990.00' ) || iso_currency_code;
```

Guidelines

- You can have the same data types for the individual columns in a multi-column use case domain as a single-column use case domain.
- For a multi-column use case domain, a column must not overlap between different domains. For example, on a table T(TC1, TC2, TC3, TC4), domains D1(C1, C2) and D2(C1, C2) cannot be associated as D1(TC1, TC2) and D2(TC2, TC3).
- Multiple ordered subsets of columns in the same table can be associated with the same domain. For example, domain D1 can be associated as D1(TC1, TC2) and D1(TC3, TC4).
- Unlike tables that can have at most one `LONG` column, domains can have multiple columns of `LONG` data type. Such domains would be useful for evaluating check conditions involving multiple `LONG` columns using the `DOMAIN_CHECK` operator.

See Also

- *Oracle Database SQL Language Reference* for the syntactic and semantic information on creating a use case domain: CREATE DOMAIN
- Specifying a Data Type for a Domain

10.1.5.2 Associating a Multi-column Use Case Domain at Table Creation

You can use the `CREATE TABLE` DDL statement to associate a multi-column use case domain with a newly created column.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-34 Associating the `US_city` Domain with Multiple Columns

You can create a `customer` table and associate the `US_city` domain with the table's three columns.

```
CREATE TABLE customer(
  cust_id  NUMBER,
  cust_name  VARCHAR2(30),
  city_name  VARCHAR2(30),
  state  VARCHAR2(2),
  zip  NUMBER,
  DOMAIN US_city(city_name, state, zip));
```

The following example returns an error because `CITY` and `STATE` columns are overlapped between domains.

```
CREATE TABLE customer(
  cust_id  NUMBER,
  cust_name  VARCHAR2(30),
  city_name  VARCHAR2(30),
  state  VARCHAR2(2),
  zip  NUMBER,
  DOMAIN US_city(city_name, state, zip),
  DOMAIN US_city(cust_name, state, zip));
```

The following example also returns an error because the `CITY_NAME` column is repeated.

```
CREATE TABLE customer(
  cust_id  NUMBER,
  cust_name  VARCHAR2(30),
  city_name  VARCHAR2(30),
  state  VARCHAR2(2),
  zip  NUMBER,
  DOMAIN US_city(city_name, city_name, zip));
```

Example 10-35 Associating the `currency` Domain with Multiple Columns

You can create an `order_items` table with its `total_paid` and `currency_code` columns associated with the `currency` domain.

```
CREATE TABLE order_items (
  order_id INTEGER, product_id INTEGER,
  total_paid NUMBER(10, 2), currency_code char (3 CHAR ),
  DOMAIN currency ( total_paid, currency_code ));
```

Guidelines

- The column names that are passed as the actual parameters to the domain must be unique.
- Domain columns can be associated with table columns with a different name.
- The `DOMAIN` keyword is mandatory.

10.1.5.2.1 Using DML on Columns Associated with Multi-column Domains

The following are some examples of DML statements that you can use on the newly created table columns with associated multi-column use case domains.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-36 Using DML Commands on the Associated Columns

Inserting values and querying the table display the results based on the `currency` domain display and order expressions.

```
INSERT INTO order_items
VALUES (1, 1,    9.99, 'USD'),
       (2, 2,    8.99, 'GBP'),
       (3, 3,    8.99, 'EUR'),
       (4, 4, 1399,    'JPY'),
       (5, 5,  825,    'INR');

SELECT order_id, product_id,
  DOMAIN_DISPLAY ( total_paid, currency_code ) amount_paid
  FROM order_items
  ORDER BY DOMAIN_ORDER ( total_paid, currency_code );
```

> **See Also**
>
> SQL Functions for Use Case Domains for details about domain functions, such as `DOMAIN_DISPLAY` and `DOMAIN_ORDER`.

The output is:

```
  ORDER_ID PRODUCT_ID AMOUNT_PAID
---------- ---------- ------------------
         3          3 EUR              8.99
         2          2 GBP              8.99
         1          1 USD              9.99
         5          5 INR            825.00
         4          4 JPY         1,399.00
```

10.1.5.3 Associating a Multi-column Use Case Domain with Existing Columns

You can use the `ALTER TABLE` DDL statement with the `MODIFY` or `ADD` clause to associate a multi-column use case domain with an existing column or a newly added column in an existing table.

Example 10-37 Associating a Multi-column Use Case Domain with Existing Columns

The following example applies the `US_city` domain to the three columns of the `customer` table.

```
ALTER TABLE customer
  MODIFY (city_name, state, zip) ADD DOMAIN US_city;
```

> **Note**
>
> The `DOMAIN` keyword is mandatory for the `ALTER TABLE .. MODIFY` statement.

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.1.5.4 Altering a Multi-column Use Case Domain

You can alter a multi-column use case domain just as you can alter a single-column use case domain. In a multi-column use case domain, you can change the `DISPLAY` and `ORDER` properties. For multi-column domains, altering annotations at the column-level is currently not supported but you can alter the object-level annotations.

Example 10-38 Altering Display and Order Expressions for a Multi-column Use Case Domain

The following `ALTER` statement changes the display expression of the `currency` domain. The current display expression shows the currency code and then the currency value. The altered display expression shows the currency value and then the currency code.

```
ALTER DOMAIN currency
  MODIFY DISPLAY TO_CHAR ( amount, '999,990.00' ) || '-' || iso_currency_code;
```

The following `ALTER` statement changes the order expression of the `currency` domain. The current order expression sorts by the currency value and then by the currency code. The altered order expression sorts by the currency code and then by the currency value.

```
ALTER DOMAIN currency
  MODIFY ORDER iso_currency_code || TO_CHAR ( amount, '999,990.00' );
```

Example 10-39 Querying the Table Associated with the Altered Multi-column Domain

```
SELECT order_id, product_id,
  DOMAIN_DISPLAY ( total_paid, currency_code ) amount_paid
  FROM order_items
  ORDER BY DOMAIN_ORDER ( total_paid, currency_code );
```

The output is:

```
ORDER_ID    PRODUCT_ID    AMOUNT_PAID
---------- ------------ --------------
         3            3       8.99-EUR
         2            2       8.99-GBP
         5            5     825.00-INR
         4            4   1,399.00-JPY
         1            1       9.99-USD
```

See Also

- *Oracle Database SQL Language Reference* for the syntactic and semantic information about altering a use case domain: ALTER DOMAIN.
- Altering a Use Case Domain

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.1.5.5 Disassociating a Multi-column Use Case Domain from a Column

You can use the `ALTER TABLE` DDL statement with the `DROP` clause to disassociate a multi-column use case domain from a column.

Example 10-40 Examples of Disassociating a Multi-column Use Case Domain

The following `ALTER TABLE` command drops the `US_City` domain from the `city_name`, `state` and `zip` columns of the `customer` table.

```
ALTER TABLE customer
  MODIFY(city_name, state, zip) DROP DOMAIN;
```

If a table T with columns (c1, c2, c3) is associated with domain D and another set of columns (c4, c5, c6) is also associated with the domain D, you can drop the domain for all the columns:

```
ALTER TABLE T
  MODIFY (c1, c2, c6, c5, c4, c3) DROP DOMAIN;
```

You cannot drop only a subset of the columns that are associated with a multi-column domain. For example, for table T, dropping only c1 and c2 columns, returns an error:

```
ALTER TABLE T
  MODIFY (c1, c2) DROP DOMAIN;
```

Example 10-41 More Examples of Disassociating a Multi-column Use Case Domain

The following code removes the `currency` domain from the `total_paid` and `currency_code` columns of the `order_items` table.

```
ALTER TABLE order_items
  MODIFY ( total_paid, currency_code ) DROP DOMAIN;
```

Guidelines

- There can be multiple ordered subsets of columns in the same table that are associated with the same domain. The removing multi-column domain syntax must specify the list of associated columns to be dissociated.
- Domain name cannot be specified.
- You cannot specify other options for `ALTER TABLE ..MODIFY` with `ALTER TABLE ..DROP DOMAIN`.

10.1.5.6 Dropping a Multi-column Use Case Domain

To drop a multi-column use case domain, use the same syntax as used for a single-column use case domain.

Guidelines

- To drop a domain that is referenced in a flexible domain, use `DROP DOMAIN` with the `FORCE` option.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for the syntactic and semantic information about dropping a use case domain: DROP DOMAIN
> - Dropping a Use Case Domain for more information about dropping a use case domain.

10.1.6 Using a Flexible Use Case Domain

This section explains how you can create, associate, disassociate, and drop a flexible use case domain.

> **Note**
>
> You cannot alter a flexible use case domain, but as an alternative, you can disassociate the flexible domain from the tables, `DROP` the domain, recreate the domain, and re-associate it with the tables.

Topics:

- Creating a Flexible Use Case Domain
- Associating a Flexible Use Case Domain at Table Creation
- Associating a Flexible Domain with Existing Columns
- Disassociating a Flexible Use Case Domain from Columns
- Dropping a Flexible Use Case Domain

10.1.6.1 Creating a Flexible Use Case Domain

You can create a flexible use case domain that references other non-flexible domains (single and multi-column use case domains) and enables you to apply one of them to table columns depending on the context of the data. For example, you can create multi-column domains to validate address formats for each country. Table columns can only belong to one domain. So, to enable the database to use the address domain corresponding to each country for each row, create a flexible domain over the country domains. Associate columns with the flexible domain using the country as the discriminant column. Each row can then apply the address rules for the corresponding country domain.

Example 10-42 Creating a Flexible Use Case Domain for Temperature Readings

The following code creates a flexible use case domain called `temperature` from three domains, namely `celcius`, `fahrenheit`, and `kelvin`. The domains that are created for each temperature scale have appropriate absolute zero checks and display expressions. There is no `ELSE` clause in the flexible domain, so you can insert values for other temperature units, and temperature values for such temperature units are unconstrained.

```
CREATE DOMAIN celcius AS NUMBER
  CONSTRAINT abs_zero_c_c CHECK ( celcius >= -273.15 )
  DISPLAY celcius || ' °C';
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CREATE DOMAIN fahrenheit AS NUMBER
  CONSTRAINT abs_zero_f_c CHECK ( fahrenheit >= -459.67 )
  DISPLAY fahrenheit || ' °F';

CREATE DOMAIN kelvin AS NUMBER
  CONSTRAINT abs_zero_k_c CHECK ( kelvin >= 0 )
  DISPLAY kelvin || ' K';
```

The following code creates a flexible domain that selects which domain to use based on the temperature units.

```
CREATE FLEXIBLE DOMAIN temperature (
  temp
 ) CHOOSE DOMAIN USING ( units char(1) )
 FROM (
  CASE units
    WHEN 'C' THEN celcius ( temp )
    WHEN 'F' THEN fahrenheit ( temp )
    WHEN 'K' THEN kelvin ( temp )
  END);
```

Example 10-43 Creating a Flexible Use Case Domain for Addresses

The following code creates multi-column use case domains to represent United States and United Kingdom (British) addresses, and a default address domain for other countries.

```
/* US addresses */
CREATE DOMAIN us_address AS (
  line_1  AS VARCHAR2(255 CHAR) NOT NULL,
  town    AS VARCHAR2(255 CHAR) NOT NULL,
  state   AS VARCHAR2(255 CHAR) NOT NULL,
  zipcode AS VARCHAR2(10 CHAR) NOT NULL
 ) CONSTRAINT us_address_c check (
   REGEXP_LIKE ( zipcode, '^[0-9]{5}(-[0-9]{4}){0,1}$' ));

/* British addresses */
CREATE DOMAIN gb_address AS (
  street   AS VARCHAR2(255 CHAR) NOT NULL,
  locality AS VARCHAR2(255 CHAR),
  town     AS VARCHAR2(255 CHAR) NOT NULL,
  postcode AS VARCHAR2(10 CHAR) NOT NULL
 ) CONSTRAINT gb_postcode_c check (
   REGEXP_LIKE (
    postcode, '^[A-Z]{1,2}[0-9][A-Z]{0,1} [0-9][A-Z]{2}$' ));

/* Default address */
CREATE DOMAIN global_address AS (
  line_1   AS VARCHAR2(255) NOT NULL,
  line_2   AS VARCHAR2(255),
  line_3   AS VARCHAR2(255),
  line_4   AS VARCHAR2(255),
  postcode AS VARCHAR2(10));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The following code creates a flexible domain that selects which multi-column address domain to use based on the country code.

```
CREATE FLEXIBLE DOMAIN address (
  line_1, line_2, line_3, line_4,
  postal_code
 )
 CHOOSE DOMAIN USING ( country_code VARCHAR2(2 char) )
 FROM (
  CASE country_code
    WHEN 'GB' THEN gb_address ( line_1, line_2, line_3, postal_code )
    WHEN 'US' THEN us_address ( line_1, line_2, line_3, postal_code )
    ELSE global_address ( line_1, line_2, line_3, line_4, postal_code )
  END);
```

> **Note**
>
> To create a flexible domain, you must have the `EXECUTE` privilege on each constituent domain.

> **See Also**
>
> *Oracle Database SQL Language Reference* for the syntactic and semantic information about creating a use case domain: CREATE DOMAIN

10.1.6.2 Associating a Flexible Use Case Domain at Table Creation

You can use the `CREATE TABLE` DDL to associate a flexible use case domain with a set of columns that are newly created by the new table. To add a flexible domain to a set of columns, specify the list of columns to be associated with the domain (in the domain column order), followed with the list of columns to be used as the discriminant (in discriminant column order in the flexible domain).

Example 10-44 Associating the `temperature` Flexible Domain with New Table Columns

Create a `sensor_readings` table using the `temperature` flexible domain, while specifying the discriminant column with the `USING` keyword.

```
CREATE TABLE sensor_readings (
  sensor_id integer, reading_timestamp TIMESTAMP,
  temperature_reading NUMBER,
  temperature_units   CHAR(1 CHAR),
  DOMAIN temperature ( temperature_reading )
    USING ( temperature_units ));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-45 Associating the `address` Flexible Domain with New Table Columns

The following code creates a new table called `addresses` and associates its columns with the `address` flexible domain.

```
CREATE TABLE addresses (
  line_1 VARCHAR2(255) NOT NULL,
  line_2 VARCHAR2(255),
  line_3 VARCHAR2(255),
  line_4 VARCHAR2(255),
  country_code   VARCHAR2(2 CHAR) NOT NULL,
  postal_code    VARCHAR2(10 CHAR),
  DOMAIN address (
    line_1, line_2, line_3, line_4, postal_code)
    USING ( country_code ));
```

> **Note**
>
> The `DOMAIN` and `USING` keywords are mandatory when associating flexible domains.

10.1.6.2.1 Using DML on Columns Associated with Flexible Domains

The following are some examples of DML statements on the newly created table columns that have associated flexible use case domains.

Example 10-46 Using DML Commands on Columns Associated with the `temperature` Flexible Domain

```
INSERT INTO sensor_readings
VALUES ( 1, timestamp'2023-06-08 12:00:00', 21.1, 'C' ),
       ( 1, timestamp'2023-06-08 12:05:00', 21.2, 'C' ),
       ( 1, timestamp'2023-06-08 12:10:00', 20.9, 'C' ),
       ( 2, timestamp'2023-06-08 12:00:00', 68.5, 'F' ),
       ( 2, timestamp'2023-06-08 12:05:00', 68.1, 'F' ),
       ( 2, timestamp'2023-06-08 12:10:00', 68.9, 'F' ),
       ( 3, timestamp'2023-06-08 12:00:00', 290.23, 'K' ),
       ( 3, timestamp'2023-06-08 12:05:00', 289.96, 'K' ),
       ( 3, timestamp'2023-06-08 12:10:00', 289.65, 'K' ),
       ( 4, timestamp'2023-06-08 12:00:00', 528.15, 'R' ),
       ( 4, timestamp'2023-06-08 12:05:00', 528.42, 'R' ),
       ( 4, timestamp'2023-06-08 12:10:00', 527.99, 'R' );

SELECT sensor_id, reading_timestamp,
  DOMAIN_DISPLAY ( temperature_reading, temperature_units ) temp
  FROM  sensor_readings;
```

> **See Also**
>
> SQL Functions for Use Case Domains for details about domain functions, such as `DOMAIN_DISPLAY` and `DOMAIN_ORDER`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The output is:

```
 SENSOR_ID READING_TIMESTAMP
TEMP
---------- ----------------------------
----------------------------------------
         1 08-JUN-2023 12.00.00.000000000 21.1
°C
         1 08-JUN-2023 12.05.00.000000000 21.2
°C
         1 08-JUN-2023 12.10.00.000000000 20.9
°C
         2 08-JUN-2023 12.00.00.000000000 68.5
°F
         2 08-JUN-2023 12.05.00.000000000 68.1
°F
         2 08-JUN-2023 12.10.00.000000000 68.9
°F
         3 08-JUN-2023 12.00.00.000000000 290.23
K
         3 08-JUN-2023 12.05.00.000000000 289.96
K
         3 08-JUN-2023 12.10.00.000000000 289.65 K
         4 08-JUN-2023 12.00.00.000000000
<null>
         4 08-JUN-2023 12.05.00.000000000
<null>
         4 08-JUN-2023 12.10.00.000000000 <null>
```

Example 10-47 Out-of-bounds Constraint Errors

The following values are out-of-bounds constraint errors for their respective temperature scales.

```
INSERT INTO sensor_readings
  VALUES ( 1, timestamp'2023-06-08 12:15:00', -400, 'C' );

INSERT INTO sensor_readings
  VALUES ( 2, timestamp'2023-06-08 12:15:00', -999, 'F' );

INSERT INTO sensor_readings
  VALUES ( 3, timestamp'2023-06-08 12:15:00',   -1, 'K' );
```

Example 10-48 Using DML on Columns Associated with the address Flexible Domain

```
-- Great Britian
INSERT INTO addresses ( line_1, line_3, country_code, postal_code )
  VALUES ( '10 Big street', 'London', 'GB', 'N1 2LA' );

-- United States
INSERT INTO addresses ( line_1, line_2, line_3, country_code, postal_code )
  VALUES ( '10 another road', 'Las Vegas', 'NV', 'US', '87654-3210' );

-- Tuvalu
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
INSERT INTO addresses ( line_1, country_code )
  VALUES ( '10 Main street', 'TV' );

SELECT * FROM addresses;
```

The output is:

```
LINE_1             LINE_2       LINE_3    LINE_4    COUNTRY_CODE
POSTAL_CODE
------------------ ------------ --------- --------- ---------------
-----------
10 Big street      <null>       London    <null>    GB              N1
2LA
10 another road    Las Vegas    NV        <null>    US
87654-3210
10 Main street     <null>       <null>    <null>    TV              <null>
```

The following `INSERT` command returns an error because it tries to insert UK address with US zip code.

```
INSERT INTO addresses ( line_1, line_3, country_code, postal_code )
  VALUES ( '10 Big street', 'London', 'GB', '12345-6789' );


ORA-11534: check constraint (schema.SYS_C0010286) due to domain constraint
schema.SYS_DOMAIN_C00639 of domain schema.ADDRESS violated
```

The following `INSERT` command returns an error because it tries to insert US address without values for the state.

```
INSERT INTO addresses ( line_1, line_2, country_code, postal_code )
  VALUES ( '10 another road', 'Las Vegas', 'US', '87654-3210' );


ORA-11534: check constraint (schema.SYS_C0010289) due to domain constraint
schema.SYS_DOMAIN_C00636 of domain schema.ADDRESS violated
```

10.1.6.3 Associating a Flexible Domain with Existing Columns

You can use the `ALTER TABLE` DDL with the `MODIFY` or `ADD` clauses to associate a flexible use case domain with an existing column or a newly added column in an existing table.

Example 10-49

The following code creates a new table called `temp_sensor_readings`.

```
CREATE TABLE temp_sensor_readings (
  sensor_id integer, reading_timestamp TIMESTAMP,
  temperature_reading NUMBER,
  temperature_units   CHAR(1 CHAR));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The following code associates the `temperature` flexible domain with an existing column called `temperature_reading`.

```
ALTER TABLE temp_sensor_readings
  MODIFY (temperature_reading, temperature_units)
  ADD DOMAIN temperature;
```

Guidelines

- The `DOMAIN` keyword is mandatory when associating flexible domains.
- The `USING` keyword is mandatory for `ALTER TABLE .. ADD` statement.
- You cannot have the same column associated with multiple flexible domains, whether as a domain column or as a discriminant column.
- You cannot have the column associated with the same domain, but with a different column positioning.

10.1.6.4 Disassociating a Flexible Use Case Domain from Columns

You can use the `ALTER TABLE` DDL with the `DROP` clause to disassociate a flexible use case domain from a column.

Example 10-50

The following code drops the `temperature` domain from the `temp_sensor_readings` table.

```
ALTER TABLE temp_sensor_readings
  MODIFY (temperature_reading, temperature_units) DROP DOMAIN;
```

Guidelines

- The domain name is not required because the database knows which columns are associated with which domains and one column can only be associated with one domain.
- You cannot specify other options for `ALTER TABLE ..MODIFY` with `ALTER TABLE ..DROP DOMAIN`.

10.1.6.5 Dropping a Flexible Use Case Domain

To drop a flexible use case domain, use the same syntax as used for a single-column use case domain.

Guidelines

- To drop a domain that is referenced in a flexible domain, use `DROP DOMAIN` with the `FORCE` option. Doing this also drops the flexible domain.
- To drop a flexible domain in the `FORCE` mode, you must have privileges to drop the constituent flexible domains.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database SQL Language Reference* for the syntactic and semantic information about dropping a use case domain: DROP DOMAIN.
- Dropping a Use Case Domain for more information about dropping a use case domain.

10.1.7 Using an Enumeration Use Case Domain

This section explains how you can create and use an enumeration use case domain (enumeration domain).

Topics:

- About Enumeration Type
- Enumeration Domains Overview
- Creating an Enumeration Domain
- Associating an Enumeration Domain at Table Creation
- Associating an Enumeration Domain with Existing Columns

10.1.7.1 About Enumeration Type

An Enumeration Type (also called ENUM or enumeration) is a data type that consists of an ordered set of values. As a language construct, ENUM can be used to define a fixed set of permitted literals (named values) of a data input field. Based on your application requirements, you can explicitly define the valid values in the column specification when creating a table. Some good examples of enumerations are days and months, or directions such as North, South, East, and West.

10.1.7.2 Enumeration Domains Overview

From release 23ai (version 23.4) onwards, Oracle Database supports the use of the enumeration type for a domain.

A domain that is created as an enumeration type is called an enumeration domain. As such, an enumeration domain defines a list of predefined values that can be stored in a column, such as a list of possible states for a customer order: open, pending, shipped, and delivered. After creating an enumeration domain, you can use the domain as the data type of a table column. Enumeration domains simplify adding enumeration type columns to tables in Oracle SQL.

The following are the general rules and guidelines for creating and using enumeration domains.

- An enumeration domain contains a set of names, and optionally, a value corresponding to a name.
- The name in an enumeration must be a valid SQL identifier.
- The names are ordinary Oracle SQL identifiers, and therefore, must obey all the restrictions that are applicable to a valid identifier in Oracle SQL.
- Every specified value must be a literal.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- The value can be of any data type that is supported for a use case domain, but all the values used in the domain must be of the same data type.
- If the values are unspecified, the value of the first name is 1. The value for each subsequent name is one more than the previous name's value.
- You can associate many names with each value.
- The names can be double quoted to bypass the keyword restrictions, to make it case sensitive, or both.
- The names inside an enumeration domain can be used wherever a literal is allowed in a scalar SQL expression.
- An enumeration domain can be used as you would any other single-column domain. However, unlike a regular domain, an enumeration domain has a default check constraint and display expression.
- An enumeration domain can be used in the `FROM` clause of the `SELECT` statement, as if it were a table.

10.1.7.3 Creating an Enumeration Domain

To create an enumeration domain, use the `CREATE DOMAIN AS ENUM` command.

> **See Also**
>
> *Oracle Database SQL Language Reference* for the syntactic and semantic information about creating a use case domain: CREATE DOMAIN

The following examples illustrate how you can create enumeration domains for various use cases.

Creating an Enumeration Domain for Order Status

The following code creates an `order_status` domain with only a set of names. The values for each name in the domain is as follows: New = 1, Open = 2, Shipped = 3, Closed = 4, and Canceled = 5.

```
CREATE DOMAIN order_status AS
 ENUM (
   New,
   Open,
   Shipped,
   Closed,
   Canceled
 );
```

Creating an Enumeration Domain with Double-quoted Names

Similar to ordinary identifiers, an enumeration name can be double quoted to bypass the keyword restrictions, to make it case sensitive, or both. The following example creates an

Copyright © 1999, 2025, Oracle and/or its affiliates.

enumeration domain called `CRUD` with the `CRUD` functions as names, but using the double quotes allows the names to bypass the keyword restrictions.

```
CREATE DOMAIN CRUD AS
 ENUM ( "Create",
        "Read",
        "Update",
        "Delete"
 );
```

Creating an Enumeration Domain for Status Codes

You can also create an enumeration domain with values assigned to names. The following domain called `Status_Code` has a status code value assigned to each state.

```
CREATE DOMAIN Status_Code AS
 ENUM (
   OK = 200,
   Not_Found = 404,
   Internal_Error = 500
 );
```

Creating an Enumeration Domain for the Days of a Week

The following code creates an enumeration domain called `Days_Of_Week` for all the days in a week. The value for each name in the domain is as follows: 0 for Sunday and Su, 1 for Monday and Mo, 2 for Tuesday and Tu, 3 for Wednesday and We, 4 for Thursday and Th, 5 for Friday and Fr, and 6 for Saturday and Sa. Therefore, each value is associated with two names.

```
CREATE DOMAIN Days_Of_Week AS
 ENUM (
   Sunday = Su = 0,
   Monday = Mo,
   Tuesday = Tu,
   Wednesday = We,
   Thursday = Th,
   Friday = Fr,
   Saturday = Sa
 );
```

Creating an Enumeration Domain for Job Titles

The following code creates an enumeration domain for the likely job titles in an organization.

```
CREATE DOMAIN Job_Title AS
 ENUM (
   Clerk = 'CLERK',
   Salesclerk = Salesperson = 'SALESCLERK',
   Manager = 'MANAGER',
   Analyst = 'ANALYST',
   President = 'PRESIDENT'
 );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Creating an Enumeration Domain for US Holidays

A value can also be a literal or a constant expression (and so it can be evaluated as part of the `CREATE DOMAIN` DDL). The following example creates an enumeration domain for US holidays with evaluated expressions.

```
CREATE DOMAIN US_Holidays_2023 AS
 ENUM (
   "New Years Day" = to_date('Jan  2, 2023', 'Mon DD, YYYY'),
   "MLK Jr. Day" = to_date('Jan 16, 2023', 'Mon DD, YYYY'),
   "Presidents Day" = to_date('Feb 20, 2023', 'Mon DD, YYYY'),
   "Memorial Day" = to_date('May 29, 2023', 'Mon DD, YYYY'),
   "Juneteenth" = to_date('Jun 19, 2023', 'Mon DD, YYYY'),
   "Independence Day" = to_date('Jul  4, 2023', 'Mon DD, YYYY'),
   "Labor Day" = to_date('Sep  4, 2023', 'Mon DD, YYYY'),
   "Columbus Day" = to_date('Sep  4, 2023', 'Mon DD, YYYY'),
   "Veterans Day" = to_date('Nov 11, 2023', 'Mon DD, YYYY'),
   "Thanksgiving Day" = to_date('Nov 23, 2023', 'Mon DD, YYYY'),
   "Christmas Day" = to_date('Dec 25, 2023', 'Mon DD, YYYY')
 );
```

Creating an Enumeration Domain for a Healthcare Application

The following is an example of an enumeration domain that a Healthcare Application might use:

```
CREATE DOMAIN Blood_Group AS
 ENUM (
   A_Positive = A_Pos = 'A +',
   A_Negative = A_Neg = 'A -',
   B_Positive = B_Pos = 'B +',
   B_Negative = B_Neg = 'B -',
   AB_Positive = AB_Pos = 'AB +',
   AB_Negative = AB_Neg = 'AB -',
   O_Positive = O_Pos = 'O +',
   O_Negative = O_Neg = 'O -'
 );
```

The following are a few more use cases for enumeration domains in a healthcare application.

- Imaging Modality: X-Ray, MRI, CT, Ultrasound.
- Prescription Frequency: Once Daily, Twice Daily, Thrice Daily, As Needed.
- Patient Condition: Stable, Critical, Recovering, Terminal.
- Allergy Type: Food, Drug, Environmental, Insect, Animal, Other.
- BMI Category: Underweight, Normal, Overweight, Obese.

10.1.7.4 Associating an Enumeration Domain at Table Creation

Similar to a single-column use case domain, you can use an enumeration domain as the data type of a table column at the time of table creation. The following example creates an `orders`

Copyright © 1999, 2025, Oracle and/or its affiliates.

table and associates the `order_status` enumeration domain (created earlier) with the `status` column.

```
CREATE TABLE orders(
 id NUMBER,
 cust VARCHAR2(100),
 status order_status
);
```

Describing the `orders` table shows that the `status` column is actually a numeric column with a single-column domain. The actual values are stored in the `status` column as numbers representing the order status.

```
DESCRIBE orders;
```

The output is:

```
Name              Null?          Type
- - - - - - - -   - - - - - - -  - - - - - - - - - - - - - - - -
ID                               NUMBER
CUST                             VARCHAR2(100)
STATUS                           NUMBER SCOTT.ORDER_STATUS
```

10.1.7.4.1 Using DML on Columns Associated with Enumeration Domains

The following are some examples of DML statements issued on the table columns that are associated with enumeration domains.

Using the `INSERT` Command on the `orders` Table

To insert data into the `orders` table, construct each row using the appropriate order status.

```
INSERT INTO orders VALUES
  (1, 'Costco', order_status.open),
  (2, 'BMW', order_status.closed),
  (3, 'Nestle', order_status.open);
```

The output is:

```
3 rows created.
```

You can list these rows out, but to see the enumeration values using their corresponding enumeration names, you can use the standard mechanism provided by domains, which is the `DOMAIN_DISPLAY()` function:

```
SELECT id, DOMAIN_DISPLAY(status) status FROM orders;
```

The output is:

```
                ID  STATUS
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
- - - - - - - - -  - - - - - - - - - -
               1  OPEN
               2  CLOSED
               3  OPEN
```

The following `SELECT` statement shows that the actual values stored in the `status` column are numbers.

```
SELECT id, status FROM orders;
```

The output is:

```
                ID   STATUS
- - - - - - - - -  - - - - -
                 1        2
                 2        4
                 3        2
```

Using the `UPDATE` Command on the `orders` Table

As with inserts, enumeration names can also be used in other DML statements, such as the `UPDATE` statement.

The following example updates the rows in the `orders` table to change the status from Closed to Canceled and Open to Closed.

```
UPDATE orders
  SET status = CASE status
                WHEN order_status.closed
                  THEN order_status.cancelled
                WHEN order_status.open
                  THEN order_status.closed
               END
  WHERE status IN(order_status.closed, order_status.open);
```

The output is:

```
2 rows updated.
```

To verify that the updates were made as expected, use the `SELECT` command with the `DOMAIN_DISPLAY` function again.

```
SELECT id, DOMAIN_DISPLAY(status) status FROM orders;
```

The output is:

```
               ID  STATUS
- - - - - - - - -  - - - - - - - - - -
                1  CLOSED
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
2  CANCELLED
3  CLOSED
```

Since the underlying data type of the `status` column is just a number, you can also directly update the status with any numeric value.

```
UPDATE orders SET status = 2 WHERE status = 5;
```

The output is:

```
1 row updated.
```

The domain check constraint verifies that it is a valid domain value:

```
UPDATE orders SET status = -7;
```

The output is:

```
ORA -11534: check constraint ... violated
```

Because enumeration names are just placeholders for literal values they can be used anywhere SQL allows literals:

```
SELECT 2*order_status.cancelled;
```

The output is:

```
2*ORDER_STATUS.CANCELLED
-  - - - - - - - - - - -
                      10
```

Using the `SELECT` Command on an Enumeration Domain

Unlike a regular domain, an enumeration domain can be treated as a table and queried using a `SELECT` statement.

```
SELECT * FROM order_status;
```

The output is:

```
ENUM_NAME         ENUM_VALUE
- - - - - - -    - - - - - -
NEW                        1
OPEN                       2
SHIPPED                    3
CLOSED                     4
CANCELLED                  5
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

However, just like a regular domain, an enumeration domain cannot be the target of a DML statement.

```
UPDATE order_status SET value = 4;
```

The output is:

```
ORA -04044: procedure , function , ... not allowed here
```

10.1.7.5 Associating an Enumeration Domain with Existing Columns

Similar to a regular domain, you can use the `ALTER-TABLE-MODIFY` command to add an enumeration domain to a column of an existing table. The following code assumes that `emp` is an existing table with `job`, `deptno`, and `comm` columns.

```
ALTER TABLE emp
 MODIFY(job)
  ADD DOMAIN Job_Title;
```

You can continue using the standard SQL statements, as follows:

```
UPDATE emp
 SET job = 'MANAGER'
  WHERE deptno = 20;
```

Additionally, you can also use an enumeration explicitly in your SQL statements, as follows:

```
UPDATE emp
 SET job = Job_Title.Salesperson
  WHERE comm IS NOT NULL;
```

10.1.8 Specifying a Data Type for a Domain

A domain data type can be one of Oracle data types. However, a qualified domain name must not collide with the qualified user-defined data types, or with Oracle built-in types.

If a column is associated with a domain, and the column data type is not specified, then the domain data type is used for the associated column as the default data type. If the associated column already has a data type, the column's data type is used.

If the domain data type is defined as non-`STRICT`, the associated column's data type only needs to be compatible with the domain data type, meaning that their data type must be the same, but the length, precision, and scale can be different. For instance, for a non-strict domain, you can associate a domain of `VARCHAR2(10)` with any `VARCHAR2` column.

If the domain data type is defined as `STRICT`, the associated column's data type must be compatible with the domain data type and also match the length, precision, and scale of the domain data type.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-51 Associating Columns with Domain Data Type

The following example creates a `year_of_birth` domain and a `email_dom` domain and associates the domains with columns to show the compatibility of domain and column data types.

```
DROP DOMAIN IF EXISTS year_of_birth;

CREATE DOMAIN year_of_birth AS NUMBER(4)
    CONSTRAINT CHECK ( (TRUNC(year_of_birth) = year_of_birth) AND
(year_of_birth >= 1900) );


DROP DOMAIN IF EXISTS email_dom;

CREATE DOMAIN email_dom AS VARCHAR2(100)
    CONSTRAINT email_chk check (REGEXP_LIKE (email_dom, '^(\S+)\@(\S+)\.
(\S+)$'));

DROP TABLE IF EXISTS newcustomers PURGE;
CREATE TABLE newcustomers (
  cust_Id NUMBER,
  cust_year_of_birth NUMBER(6) DOMAIN year_of_birth,
  cust_hq_email VARCHAR2(50),
  cust_office_email VARCHAR2(100),
  cust_rep_email VARCHAR2(100));

DESC newcustomers;
```

The output is:

```
 Name                          Null?    Type
 ----------------------------- -------- ----------------------------
 CUST_ID                                NUMBER
 CUST_YEAR_OF_BIRTH                     NUMBER(6) SH.YEAR_OF_BIRTH
 CUST_HQ_EMAIL                          VARCHAR2(50)
 CUST_OFFICE_EMAIL                      VARCHAR2(100)
 CUST_REP_EMAIL                         VARCHAR2(200)
```

The `cust_year_of_birth` column is defined as an Oracle data type: `Number`, and also associated with the `year_of_birth` domain, so the column's data type is assigned to the column. The `cust_year_of_birth` column inherits all the properties defined in the `year_of_birth` domain, such as constraint, display, and ordering properties.

The following example creates a `ukcustomers` table with a column associated with the `year_of_birth` domain, but without the column's data type:

```
CREATE TABLE ukcustomers (
  cust_Id NUMBER,
  cust_year_of_birth DOMAIN year_of_birth);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
DESC ukcustomers;
```

The output is:

```
 Name                          Null?    Type
 ----------------------------- -------- ----------------------------
 CUST_ID                                NUMBER
 CUST_YEAR_OF_BIRTH                     NUMBER(4) SH.YEAR_OF_BIRTH
```

Here, the `cust_year_of_birth` column is assigned the domain's data type, which is `NUMBER(4)`.

In the following example, the `DOMAIN` keyword is omitted.

```
CREATE TABLE incustomers (
  cust_id NUMBER,
  cust_year_of_birth year_of_birth);

DESC incustomers;
```

The output is:

```
 Name                          Null?    Type
 ----------------------------- -------- ----------------------------
 CUST_ID                                NUMBER
 CUST_YEAR_OF_BIRTH                     NUMBER(4) SH.YEAR_OF_BIRTH
```

In the column definition clause, the domain clause must either replace the data type clause, or immediately follow it.

If a domain column data type is not defined as `STRICT`, you can associate a domain to any column with the same data type, irrespective of the column length.

The following `ALTER` commands succeed because the domain and column data type has the same data type and the column lengths are not checked for non-`STRICT` domains.

```
ALTER TABLE newcustomers
  MODIFY cust_hq_email DOMAIN email_dom;

ALTER TABLE newcustomers
  MODIFY cust_office_email DOMAIN email_dom;

ALTER TABLE newcustomers
  MODIFY cust_rep_email DOMAIN email_dom;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

If a domain column data type is defined as STRICT, the domain association works only when the column and the domain have the same data type and their lengths also match.

```
DROP DOMAIN IF EXISTS email_dom;

CREATE DOMAIN email_dom AS VARCHAR2(100) STRICT
  CONSTRAINT email_chk check (REGEXP_LIKE (email_dom, '^(\S+)\@(\S+)\.
(\S+)$'));
```

The following ALTER command succeeds.

```
ALTER TABLE newcustomers
  MODIFY cust_office_email DOMAIN email_dom;
```

The following ALTER commands fail because the column length and the domain length do not match.

```
ALTER TABLE newcustomers
  MODIFY cust_hq_email DOMAIN email_dom;

ALTER TABLE newcustomers
  MODIFY cust_rep_email DOMAIN email_dom;
```

Table Column Data Type to non-STRICT Domain Column Data Type Compatibility

Each of the following points lists the compatible types. You can associate any table column with a domain column that has a compatible type. The table and domain columns can have different lengths, precisions, and scales for non-STRICT domains.

- NUMBER, NUMER(p), NUMBER(p, s), NUMERIC, NUMERIC(p), NUMERIC(p, s), DECIMAL, DECIMAL(p), DEC, DEC(p), INTEGER, INT, SMALLINT, FLOAT, FLOAT(p), REAL, DOUBLE_PRECISION
- CHAR(L), CHAR(L CHAR), CHAR(L BYTE), CHARACTER(L CHAR), CHARACTER(L BYTE), CHARACTER(L)
- NCHAR(L), NATIONAL CHARACTER(L), NATIONAL CHAR (L)
- VARCHAR2(L), VARCHAR2(L CHAR), VARCHAR2(L BYTE), CHAR VARYING(L CHAR), CHAR VARYING(L BYTE), CHAR VARYING(L), CHARACTER VARYING(L CHAR), CHARACTER VARYING(L BYTE), CHARACTER VARYING(L)
- NVARCHAR2(L), NATIONAL CHAR VARYING (L), NATIONAL CHARACTER VARYING (L)
- TIMESTAMP, TIMESTAMP(p)
- TIMESTAMP WITH TIME ZONE, TIMESTAMP(p) WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE, TIMESTAMP(p) WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH, INTERVAL YEAR(p) TO MONTH
- INTERVAL DAY TO SECOND, INTERVAL DAY(p) TO SECOND, INTERVAL DAY TO SECOND(s), INTERVAL DAY(p) TO SECOND(s)
- ROWID, UROWID, UROWID(p)
- RAW(p), RAW(p)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table Column Data Type to STRICT Domain Column Data Type Compatibility

Each of the following points lists the compatible types. You can associate any table column with a domain column that has a compatible type. The table and domain columns must have an exact match for length, precision, and scale for STRICT domains.

- NUMBER(*), NUMBER
- NUMBER(p, 0), NUMERIC(p), NUMERIC(p, 0), DECIMAL(p), DEC(p), provided the table column data type and the domain column data type have the same precision.
- NUMBER(p, s), NUMERIC(p, s), DECIMAL(p, s), DEC(p, s), provided the table column data type and the domain column data type have the same precision and scale.
- NUMBER(*,0), NUMERIC(*), NUMERIC(*,0), DEC(*), DEC(*,0), DECIMAL(*), DECIMAL(*,0), INTEGER, INT, SMALLINT
- FLOAT(63), REAL
- FLOAT, FLOAT(126), DOUBLE PRECISION
- CHAR(L CHAR), CHAR(L BYTE), CHARACTER (L), provided the size in bytes is the same for the column data type and domain column data type. For example, CHAR (4 BYTE) can be associated with a STRICT domain column of CHAR(1 CHAR) if 1 character takes 4 bytes.
- NCHAR(L), NATIONAL CHARACTER(L), NATIONAL CHAR (L), provided the size in bytes is the same for the column data type and domain column data type.
- VARCHAR2(L CHAR), VARCHAR2(L BYTE), CHARACTER VARYING (L), CHAR VARYING(L), provided the size in bytes is the same for the column data type and domain column data type.
- NVARCHAR2(L), NATIONAL CHAR VARYING (L), NATIONAL CHARACTER VARYING (L), provided the size in bytes is the same for the column data type and domain column data type.
- TIMESTAMP, TIMESTAMP(6)
- TIMESTAMP WITH TIME ZONE, TIMESTAMP(6) WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE, TIMESTAMP(6) WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH, INTERVAL YEAR(2) TO MONTH
- INTERVAL DAY TO SECOND, INTERVAL DAY(2) TO SECOND, INTERVAL DAY TO SECOND(6), INTERVAL DAY(2) TO SECOND(6)
- ROWID, UROWID(10)
- UROWID, UROWID(4000), ROWID(4000)

Table Column Data Type to Domain Column Data Type Compatibility

The following column data types are only compatible with an equivalent domain column data type. For example, a DATE column type can be associated only with a DATE domain column type.

- BINARY_FLOAT
- BINARY_DOUBLE
- DATE
- BLOB
- CLOB

Copyright © 1999, 2025, Oracle and/or its affiliates.

- NCLOB
- BFILE
- LONG
- LONG RAW
- JSON

Rules of Associating Table Column and Use Case Domain Based on Data Type

For a domain's data type of VARCHAR2(L [CHAR|BYTE]), let L-bytes be the maximum length in bytes corresponding to L, given that the National Language Support (NLS) setting has the session-level length semantics value in NLS_LENGTH_SEMANTICS as BYTE.

The following rules apply when you associate a column with the domain:

- If the domain is defined as non-STRICT, the domain can be associated with columns of data type VARCHAR2(x) for any x-bytes. For non-STRICT domains, L and x can be different.
- If the domain is defined as STRICT, the domain can be associated with columns of data type VARCHAR2(x) for any x-bytes = L-bytes. For STRICT domains, L and x must be the same number of bytes, even after converting L|x CHAR to BYTES, if needed.

For instance, if a domain data type specification is VARCHAR2 STRICT, and if MAX_STRING_SIZE is STANDARD, then the domain can associate with columns of VARCHAR2(L BYTE) data type, where L = 4000. If the current NLS settings in the session are such that at most 2 bytes are needed to represent a character, then a column of VARCHAR2(2000 CHAR) data type can be associated with the domain. If MAX_STRING_SEMANTICS is changed to EXTENDED, then columns of data type: VARCHAR2(L BYTE) for L = 32767 or VARCHAR2(16383 CHAR) can be associated with the domain.

Similar rules apply to NVARCHAR2(L), CHAR(L [CHAR|BYTE]), and NCHAR(L).

10.1.9 Changing the Use Case Domain Properties

As your application evolves, column definitions may need to change, and with that the properties of the associated use case domains must also change. At the time of altering a domain, you can only alter the annotations, display expression, and order expression for domains. The following suggests ways to change the other domain properties.

You can use these examples as a means to get started, or to build upon and adapt them to your specific data and business requirements. The first example is a manual method and the next one is an online method using the dbms_redefinition package.

Example 10-52 Changing the Domain Properties Manually

Suppose that you want to change the string length of an email domain field from 100 to 200. To make that change, you must complete the following steps:

- Alter the associated column's string length to 200.
- Drop the current domain associated with the column.
- Create a new domain with the email string length as 200.
- Re-associate columns associated with the current domain with the new domain.

If the current domain definition and column association are as follows:

```
CREATE DOMAIN email_dom AS VARCHAR2(100)
  CONSTRAINT email_chk CHECK (regexp_like (email_dom, '^(\S+)\@(\S+)\.(\S+)$'));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CREATE TABLE t1 (
  id NUMBER,
  email DOMAIN email_dom
);
```

To view the list of columns associated with the domain:

```
SELECT table_name, column_name
  FROM user_tab_columns
  WHERE domain_name = 'EMAIL_DOM';

TABLE_NAME              COLUMN_NAME
----------------------- -------------------
T1                      EMAIL
```

Alter the t1 table to change the email column with string size as 200:

```
ALTER TABLE t1 modify email varchar2(200);
```

Drop the domain with the `FORCE PRESERVE` option:

```
DROP DOMAIN email_dom FORCE PRESERVE;
```

Then, create a new domain with the email string size as 200:

```
CREATE DOMAIN email_dom AS VARCHAR2(200)
  CONSTRAINT email_chk CHECK (regexp_like (email_dom, '^(\S+)\@(\S+)\.(\S+)$'));
```

And, re-associate the new domain with the email column:

```
ALTER TABLE t1
  MODIFY email DOMAIN email_dom;
```

Example 10-53 Changing the Domain Properties Using the online `dbms_redefinition` Package

You can use the Oracle online table reorganization package called `dbms_redefinition` to change permitted values, such as updating which currencies are supported. For instance, to change the supported currencies in a currency domain, the required steps are:

- Create a new domain.
- Migrate columns associated with the current domain to use the new domain.

The following example has only basic error handling. There can be instances where an error can occur after several tables are migrated, such as when some data violates the new constraint. A complete solution would need to account for these scenarios.

The following code creates a domain with a constraint that allows the following currencies: USD, GBP, and EUR.

```
CREATE DOMAIN currency AS (
  amount AS NUMBER,
  iso_currency_code AS CHAR ( 3 CHAR )
 ) CONSTRAINT supported_currencies_c
    CHECK ( iso_currency_code in ( 'USD', 'GBP', 'EUR' ) );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Suppose that the following tables have columns associated with the `currency` domain.

```
CREATE TABLE order_items (
  order_id   INTEGER, product_id INTEGER,
  total_paid NUMBER(10, 2), currency_code CHAR ( 3 CHAR ),
  DOMAIN currency ( total_paid, currency_code ),
  PRIMARY KEY ( order_id, product_id )
);

CREATE TABLE product_prices (
  product_id    INTEGER,
  unit_price    NUMBER(10, 2),
  currency_code CHAR( 3 char ),
  DOMAIN currency ( unit_price, currency_code ),
  PRIMARY KEY ( product_id, currency_code )
);
```

Use the `INSERT` command to store some values into the `product_prices` and `order_items` tables.

```
INSERT INTO product_prices
VALUES (1, 9.99, 'USD'),
       (1, 8.99, 'GBP'),
       (1, 8.99, 'EUR');

INSERT INTO order_items
VALUES (1, 1, 9.99, 'USD'),
       (2, 1, 8.99, 'GBP'),
       (3, 1, 8.99, 'EUR');

COMMIT;
```

Suppose that your business is expanding and you want to support more currencies. You cannot modify the constraints directly, so you need an alternative approach that includes:

- Creating a new domain.
- Altering the `product_prices` and `order_items` tables to drop the existing domain from the associated columns while preserving the constraints.
- Altering the tables to add the new domain.
- Removing the preserved constraints from the tables.

However, when done online, these are blocking DDL statements. Instead, you can use the `dbms_redefinition` package to modify the constraints online.

You must create a new domain with the constraint including the newly supported currencies, and associate it with temporary tables. The new domain replaces the original domain.

```
CREATE DOMAIN currency_d as (
  amount            AS NUMBER,
  iso_currency_code AS CHAR ( 3 CHAR )
 ) CONSTRAINT supported_currencies_d_c
  CHECK ( iso_currency_code in ( 'USD', 'GBP', 'EUR', 'CAD', 'MXP', 'INR',
'JPY' ) );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CREATE TABLE order_items_tmp (
  order_id   INTEGER, product_id INTEGER,
  total_paid NUMBER(10, 2), currency_code CHAR ( 3 CHAR ),
  DOMAIN currency_d ( total_paid, currency_code ) ,
  PRIMARY KEY ( order_id, product_id )
);

CREATE TABLE product_prices_tmp (
  product_id    INTEGER,
  unit_price    NUMBER(10, 2),
  currency_code CHAR (3 CHAR),
  DOMAIN currency_d ( unit_price, currency_code ),
  PRIMARY KEY ( product_id, currency_code )
);
```

To make the code more reusable, you can create a `redefine_table` procedure that calls the `dbms_redefinition` procedures to copy the properties of the temporary table columns to the current table columns, and then swap the current table columns to the new domain.

```
DECLARE

  PROCEDURE redefine_table ( current_table VARCHAR2, staging_table VARCHAR2 )
AS
    num_errors pls_integer;
  BEGIN
    DBMS_REDEFINITION.CAN_REDEF_TABLE(user, current_table);
    DBMS_REDEFINITION.START_REDEF_TABLE(user, current_table, staging_table);

    DBMS_REDEFINITION.copy_table_dependents(
      uname               => user,
      orig_table          => current_table,
      int_table           => staging_table,
      copy_constraints    => false,
      num_errors          => num_errors);

    IF num_errors > 0 THEN
      dbms_redefinition.abort_redef_table(user, current_table, staging_table);
      raise_application_error ( -20001, num_errors || ' errors copying
dependents from ' || current_table || ' to ' || staging_table );
    ELSE
      dbms_redefinition.finish_redef_table(user, current_table,
staging_table);
    END IF;
  END redefine_table;

BEGIN

  FOR tabs IN (
    SELECT distinct table_name from user_tab_cols
    WHERE  domain_name = 'CURRENCY'
  ) LOOP
    redefine_table(tabs.table_name, tabs.table_name || '_TMP');
  END LOOP;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
END;
/
```

Use DML on the `product_prices` table to see if the new currencies are now supported.

```
-- New currencies now supported
INSERT INTO product_prices
  VALUES (1, 9.99, 'CAD');

-- Invalid currencies raise exception
INSERT INTO product_prices
  VALUES (1, 9.99, 'N/A');

SELECT * FROM product_prices;
```

The output is:

```
PRODUCT_ID UNIT_PRICE CUR
---------- ---------- ---
         1       9.99 USD
         1       8.99 GBP
         1       8.99 EUR
         1       9.99 CAD
```

Clean up the temporary objects and old domain: `currency`.

```
DROP TABLE order_items_tmp PURGE;
DROP TABLE product_prices_tmp PURGE;
DROP DOMAIN currency;
```

> **See Also**
>
> DBMS_REDEFINITION in *PL/SQL Packages and Types Reference* Guide.

10.1.10 SQL Functions for Use Case Domains

Domain functions enable you to work with use case domains more efficiently.

You can use the following SQL functions with use case domains:

- `DOMAIN_DISPLAY` returns the domain display expression for the domain that the argument is associated with.
- `DOMAIN_NAME` returns the qualified domain name of the domain that the argument is associated with. Note that if there is a public synonym to the domain, it is returned; otherwise domain name is returned in `domain_owner.domain_name` format.
- `DOMAIN_ORDER` returns the domains order expression for the domain that the argument is associated with.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `DOMAIN_CHECK(domain_name, value1, value2, ...)` applies constraint conditions of `domain_name` (not null or check constraint) to the `value` expression. It also checks the values for type compatibility. There can be many values; the number of value expressions must match the number of columns in the domain or the statement raises an error.
- `DOMAIN_CHECK_TYPE(domain_name, value1, value2, ...)` verifies whether the input expressions are compatible with the domain's data type. There can be many values; the number of value expressions must match the number of columns in the domain or the statement raises an error.

> **See Also**
>
> Domain Functions in *Oracle Database SQL Language Reference* for more information about using SQL functions for use case domains

10.1.11 Viewing Domain Information

You can use the dictionary views to get information about the domains. Dictionary views can also help identify columns that have different properties, such as constraint and default expressions when compared to their associated domain.

10.1.11.1 Dictionary Views for Use Case Domains

Dictionary views: `[USER|DBA|ALL]_DOMAINS` and `[USER|DBA|ALL]_DOMAIN_COLS` represent domains and provide the following information about the domain columns. For flexible domains, the views also include the domain selector expression.

- Domain name
- Domain owner
- Display and ordering expression
- Default value
- Data type of the domain
- Collation

The following dictionary views are available for use case domains:

- `ALL_DOMAINS` describes the domains accessible to the current user.
- `DBA_DOMAINS` describes all domains in the database.
- `USER_DOMAINS` describes the domains owned by the current user.
- `ALL_DOMAIN_COLS` describes columns of the domains accessible to the current user.
- `DBA_DOMAIN_COLS` describes columns of all domains in the database.
- `USER_DOMAIN_COLS` describes columns of the domains owned by the current user.
- `ALL_DOMAIN_CONSTRAINTS` describes constraint definitions in the domains accessible to the current user.
- `DBA_DOMAIN_CONSTRAINTS` describes constraint definitions in all domains in the database.
- `USER_DOMAIN_CONSTRAINTS` describes constraint definitions in the domains owned by the current user.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database Reference* for more information about the following views that are used for use case domains: ALL_DOMAINS, DBA_DOMAINS, USER_DOMAINS, ALL_DOMAIN_COLS, DBA_DOMAIN_COLS, USER_DOMAIN_COLS, ALL_DOMAIN_CONSTRAINTS, DBA_DOMAIN_CONSTRAINTS, USER_DOMAIN_CONSTRAINTS

10.1.12 Built-in Use Case Domains

Oracle Database provides some built-in use case domains that you can use directly on table columns, for example, `email_d`, `ssn_d`, and `credit_card_number_d`. Built-in use case domains exist in all PDBs. When a new PDB is added into a CDB, the built-in use case domains automatically get created and added in the PDB.

The following built-in domains are supported in Oracle Database. These are categorized as follows:

- Identifier Built-in Domains
- Tech Built-in Domains
- Numeric Built-in Domains
- Miscellaneous Built-in Domains

Table 10-1 Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
1	`city_d`	Urban area or municipality	"San Francisco", "Mumbai", "Tokyo"	VARCHAR2(100)	address	-	-	-
2	`country_code_d`	A standardized two-letter or three-letter code assigned to each country or territory, typically defined by international standards such as ISO 3166.	"USA", "IND", "AUS"	VARCHAR2(3)	address	-	-	-
3	`country_d`	A distinct territorial and sovereign entity recognized as an independent nation-state, characterized by its own government, laws, and international recognition.	"USA", "India", "Mexico"	VARCHAR2(100)	address	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
4	credit_card_number_d	A unique series of digits assigned to a credit card for identification and transaction purposes.	"123456789012345" "123456789012345" "123456789012345" "123456789012345" "1234567890123456" "123456789012345678"	VARCHAR2(20)	-	REGEX - ^[0-9]{12,19}$	-	WHEN length(credit_card_number_d)=15 THEN SUBSTR(credit_card_number_d,1,4) || ' ' || SUBSTR(credit_card_number_d,5,6) || ' ' || SUBSTR(credit_ca

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
								rd_number_d, 11,5) WHEN length(credit_card_number_d)=16 THEN SUBSTR(credit_card_number_d, 1,4) || ' ' || SUBSTR(credit_card_number_d, 5,4) || ' ' ||

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
								SUBSTR(credit_card_number_d,9,4) || ' ' || SUBSTR(credit_card_number_d,13,4) WHEN length(credit_card_number_d)=14 THEN SUBSTR(credit_card_number_d,1,4) || ' '

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display

								`||`
								
								`SUBSTR(credit_card_number_d, 5,6) || ' ' || SUBSTR(credit_card_number_d, 11,4)`

5	`date_of_birth_d`	The specific day, month, and year on which an individual was born, representing their birth date. It is a fundamental piece of personal information used for identification, age verification, and record-keeping purposes. The displayed format depends on the date format used by the session.	01-JAN-91, 02/28/2023, 2023/5 4	DATE	person_info	-	-	-
6	`district_d`	A defined area or region within a city or country, often characterized by distinct administrative, social, or geographical features.	"Manhattan", "Brooklyn"	VARCHAR2(100)	address	-	-	-
7	`floor_d`	The level or story within a building where a specific unit or residence is situated, often indicated in its address.	"1", "1A", "Ground", "First"	VARCHAR2(20)	address	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
8	house_number_d	The numerical identifier assigned to a building or residence within a street or locality, typically used for postal addressing and navigation purposes.	"1-A", "10", "C1-H123"	VARCHAR2(50)	address	-	-	-
9	license_plate_d	A unique combination of letters, numbers, or symbols displayed on a vehicle's license plate, serving as a unique identifier for that vehicle.	"ABC123", "1234 XYZ", "DEF-456", "789 GHI", "JKL 987", "MNO-654", "PQR1234", "STU 5678", "VWX-9012", "YZA 3456"	VARCHAR2(100)	person_info	-	-	-
10	marital_status_d	A demographic attribute indicating an individual's legal relationship status with respect to marriage. Common categories include "single," "married," "divorced," "widowed," or "separated", among others.	"single", "married", "widowed"	VARCHAR2(50)	person_info	-	-	-
11	national_id_d	A unique identifier issued by a government to its citizens or residents for identification and administrative purposes. National IDs are used to access government services, prove identity, and facilitate transactions such as voting, travel, and financial activities.	"123-45-6789", "AB123456C", "123456789012", "123.456.789-01"	VARCHAR2(50)	person_info	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
12	passport_d	A government-issued travel document that certifies the holder's identity and nationality, allowing them to travel internationally.	"AB123456","123456789","ABC123456","E1234567","AB1234567","123456789AB"	VARCHAR2(100)	person_info	-	-	-
13	payment_card_id_d	A unique identifier associated with a payment card, such as a credit card, debit card, or prepaid card. This identifier distinguishes one card from another within a financial institution's system and is used for authorization, processing, and tracking of transactions.	"4012 8888 8888 1881","6011 1111 1111 1117","3714 4963 5398 431","5555 5555 5555 4444","5105 1051 0510 5100"	VARCHAR2(200)	person_info, payment_info	-	-	-
14	phone_d	A numerical sequence used for telecommunications, allowing individuals to connect through voice calls or text messages.	"1-555-555-5555", "44-20-1234-5678", "61-2-1234-5678", "49-30-12345678"	VARCHAR2(20)	person_info	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
15	`phone_number_d`	A numerical sequence used for telecommunications, allowing individuals to connect through voice calls or text messages.	"+1234567890123456" "1234567890 12345" "+9876543210 98765" "98765432109 8765" "+123" "123"	VARCHAR2(17)	-	REGEX - ^[+]{0,1}[0-9]{1,16}$	-	-
16	`postal_code_d`	A numerical or alphanumeric code used by postal services to identify specific geographic areas for mail delivery and sorting.	"SW1A 1AA", "M5H 2N2", "2000", "10117", "75001", "100-0001", "22010-000", "100000", "110001", "2000"	VARCHAR2(20)	address	-	-	-
17	`province_d`	A territorial division within a country, typically possessing its own government and administrative authority, particularly in federal or decentralized systems.	"Alberta", "Ontario"	VARCHAR2(100)	address	-	-	-
18	`social_media_id_d`	A unique identifier associated with an individual or entity's account on a specific social media platform. It serves as a distinct label or reference point within the platform's system, allowing for identification and interaction with specific users or profiles. This domain is typically used in conjunction with `social_media_type_d` domain that represents the social media platform.	"user123", "nyname@email.com", "myaccount11", "203"	VARCHAR2(100)	person_info	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
19	`social_media_type_d`	Refers to the classification or category of a social media platform, such as Facebook, Twitter, Instagram, or LinkedIn. This domain is typically used in conjunction with `social_media_id_d` domain that represents the user identifier of a social media platform.	"facebook", "x", "instagram"	VARCHAR2(100)	-	-	-	-
20	`ssn_d`	A unique nine-digit identifier assigned by the Social Security Administration to individuals in the United States.	"123-45-6789", "987-65-4321", "456-78-9012", "876-54-3210", "234-56-7890"	VARCHAR2(11)	-	REGEX - ^[0-9]{3}[-][0-9]{2}[-][0-9]{4}$	-	-
21	`state_d`	A territorial and administrative division within a country, often possessing its own government and legal system.	"California", "Texas"	VARCHAR2(100)	address	-	-	-
22	`street_d`	The name of a road or thoroughfare where a building or residence is located, forming part of its address.	"4th Main Road", "Purple Street 104"	VARCHAR2(200)	address	-	-	-
23	`us_license_plate_d`	A unique alphanumeric combination displayed on a vehicle's license plate in the United States, serving as a distinctive identifier for that vehicle.	"ABC-XY-1234", "DEF-GH-567", "JKL-MN-890", "OPQ-RS-4567", "TUV-WX-8901"	VARCHAR2(20)	person_info	REGEX - ^[A-Za-z]{1,3}-[A-Za-z]{1,2}-[0-9]{1,4}$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
24	us_passport_d	A travel document issued by the United States government to its citizens for international travel.	"ABC123456", "XYZ789012", "DEF456789", "GHI012345", "12345678"	VARCHAR2(100)	person_info	-	-	-
25	us_phone_d	A series of digits used for telecommunication in the United States.	"123-456-7890", "1234567890", "(123) 456-7890", "(123)456-7890", "123.456.7890", "1234567890"	VARCHAR2(20)	person_info	REGEX - ^\D?(\d{3})\D?\D?(\d{3})\D?(\d{4})$	-	-
26	us_postal_code_d	United States postal code, typically referring to ZIP codes.	"10001-1234", "90210-5678", "60601-9876", "02110-5432", "33109-8765"	VARCHAR2(20)	address	REGEX - ^\d{5}([\-]\d{4})?$	-	-
27	us_ssn_d	A unique nine-digit identifier assigned by the Social Security Administration to individuals in the United States.	"123-45-6789", "123456789"	VARCHAR2(15)	person_info	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
28	`us_state_d`	Administrative region within the United States.	"CA", "NY", "TX", "FL", "WA", "OH", "PA", "IL", "GA", "MI", "VA", "NC", "NJ", "MD", "AZ", "CO", "MA", "TN", "IN", "MO"	VARCHAR2(2)	address	REGEX - ^(AE|AL|AK|AP|AS|AZ|AR|CA|CO|CT|DE|DC|FM|FL|GA|GU|HI|ID|IL|IN|IA|KS|KY|LA|ME|MH|MD|MA|MI|MN|MS|MO|MP|MT|NE|NV|NH|NJ|NM|NY|NC|ND|OH|OK|OR|PW|PA|PR|RI|SC|SD|TN|TX|UT|VT|VI|VA|WA|WV|WI|WY)$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
29	`us_vehicle_id_d`	A unique identifier assigned to a vehicle in the United States, often referred to as a Vehicle Identification Number (VIN).	"1G1JC1441Y7167030", "5XYZ1234567890123", "WBAFA53541LM82357", "2G1WG5EK5B1160036", "3N1AB7AP2JY320999"	VARCHAR2(20)	person_info	REGEX - ^[A-HJ-NPR-Z0-9]{17}$	-	-
30	`us_zip_d`	An alphanumeric code used by the United States Postal Service (USPS) to facilitate mail delivery and address sorting.	"10001-1234", "90210-5678", "60601-9876", "02110-5432", "33109-8765"	VARCHAR2(15)	address	REGEX - ^\d{5}(-\d{4})?$	-	-
31	`vehicle_id_d`	A unique identifier assigned to a vehicle, typically used for registration, tracking, and identification purposes by government authorities and transportation agencies.	"UK11AP999"	VARCHAR2(100)	person_info	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-1 (Cont.) Identifier Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
32	zip_d	A postal code used by postal services to efficiently route mail to specific geographic regions.	"SW1A 1AA", "M5H 2N2", "2000", "10117", "75001", "100-0001", "22010-000", "100000", "110001", "2000"	VARCHAR2(20)	address	-	-	-

Table 10-2 Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
1	cidr_d	CIDR: Abbreviation for Classless Inter-Domain Routing, used for specifying the size of an IP network.	"192.168.1.0/24", "10.0.0.0/8"	VARCHAR2(18)	tech_info	REGEX - ^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\/([1-9]|[1-2][0-9]|3[0-2])$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
2	`created_on_d`	Date and time when a file, record, or entity was created.	'1960-01-01 23:03:20'	TIMESTAMP	tech_info, timestamp	-	-	-
3	`description_d`	Textual information providing details or characteristics about something.	"Vintage red bicycle", "Cozy wooden cabin", "Sparkling crystal chandelier", "Rustic farmhouse kitchen", "Gentle ocean breeze"	VARCHAR2(4000)	-	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
4	email_d	Electronic mail address used for digital communication.	"example@example.com", "user123@gmail.com"	VARCHAR2(4000)	person_info	REGEX - ^([a-zA-Z0-9!#$%&*+=?^_`{\|}~-]+(\.[A-Za-z0-9!#$%&*+=?^_`{|}~-]+)*)@((([a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?\.)+[a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?)$	-	-
5	encryption_function_d	Algorithm used to encode data for secure transmission or storage.	"AES", "DES", "RSA", "Blowfish", "Twofish", "RC4"	VARCHAR2(1000)	tech_info, encryption	-	-	-
6	encryption_value_d	Data transformed using encryption algorithms to conceal its meaning. This domain is typically used in conjunction with encryption_function_d domain that represents the encryption function that was used.	"5d41402abc4b2a76b9719d911017c592"	VARCHAR2(1000)	tech_info, encryption	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
7	file_extension_d	Suffix added to the name of a computer file to denote its format or usage.	.txt", ".jpg", ".pdf", ".mp3"	VARCHAR2(10)	tech_info	-	-	-
8	file_size_d	Magnitude of data in a computer file as measured in an arbitrary unit, but typically measured in bytes.	100, 500, 1000, 2000, 5000	NUMBER	tech_info	CHECK >=0	-	-
9	hash_function_d	Algorithm used to map data of arbitrary size to fixed-size values.	"MD5", "SHA-1", "SHA-256", "SHA-512", "CRC32"	VARCHAR2(1000)	tech_info, hash	-	-	-
10	hash_value_d	Result of applying a hash function to a data set, typically used for data integrity verification. This domain is typically used in conjunction with hash_function_d domain that represents the hash function which was used.	"5d41402abc4b2a76b9719d911017c592"	VARCHAR2(1000)	tech_info, hash	-	-	-
11	hostname_d	Unique label assigned to a device within a network.	"example.com" "subdomain.example.com" "mail.example.com" "123example456.com"	VARCHAR2(255)	tech_info	REGEX - ^([a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9\-]{0,61}[a-zA-Z0-9])(\.([a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9\-]{0,61}[a-zA-Z0-9]))*$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
12	`id_number_d`	Unique identifier assigned to an individual, entity, or object. An equivalent domain: `id_string_d` can be used if the identifier is not numeric.	9876543210, 2468135790, 1357924680, 5555555555	NUMBER	-	-	-	-
13	`id_string_d`	Sequence of characters used to uniquely identify an entity. An equivalent domain: `id_number_d` can be used if the identifier is numeric.	"abc123DEF456"	VARCHAR2(4000)	-	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
14	`ip_address_d`	Unique numerical label assigned to devices participating in a computer network.	"172.16.0.1", "255.255.255.0", "2001:0db8:85a3:0000:0000:8a2e:0370:7334"	VARCHAR2(46)	tech_info	REGEX - ^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$ or ^([0-9a-fA-F]{4}[:]){7}[0-9a-fA-F]{4}$ or ^([0-9a-fA-F]{4}[:]){6}(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
15	ipv4_address_d	Numeric label assigned to devices participating in a computer network using the Internet Protocol version 4.	"192.168.0.1", "10.0.0.1", "172.16.0.1", "255.255.255.0", "127.0.0.1", "8.8.8.8", "172.31.255.255", "198.51.100.1"	VARCHAR2(15)	tech_info	REGEX - ^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$	-	-
16	ipv6_address_d	Numerical label assigned to devices participating in a computer network using the Internet Protocol version 6.	"2001:0db8:85a3:0000:0000:8a2e:0370:7334"	VARCHAR2(46)	tech_info	REGEX - ^([0-9a-fA-F]{4}[:]){7}[0-9a-fA-F]{4}$ or ^([0-9a-fA-F]{4}[:]){6}(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
17	`last_modified_on_d`	Date and time when a file or resource was last modified.	'1960-01-01 23:03:20'	TIMESTAMP	tech_info, timestamp	-	-	-
18	`last_opened_on_d`	Date and time when data or information was last accessed.	'1960-01-01 23:03:20'	TIMESTAMP	tech_info, timestamp	-	-	-
19	`last_updated_on_d`	Date and time when a file or resource was last opened.	'1960-01-01 23:03:20'	TIMESTAMP	tech_info, timestamp	-	-	-
20	`mac_address_d`	Unique identifier assigned to a network interface controller for communications within a network.	"00-1A-2B-3C-4D" "FF-A1-B2-C3-D4" "ab-cd-ef-12-34" "00:1A:2B:3C:4D" "FF:A1:B2:C3:D4" "ab:cd:ef:12:34" "001A.2B3C.4D5E" "FFA1.B2C3.D4E5" "abcd.ef12.3456"	VARCHAR2(17)	tech_info	REGEX - ^([a-fA-F0-9]{2}[-]){5}[a-fA-F0-9]{2}$ or ^([a-fA-F0-9]{2}[:]){5}[a-fA-F0-9]{2}$ or ^([a-fA-F0-9]{4}[.]){2}[a-fA-F0-9]{4}$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
21	mime_type_d	Identifier for the type of data transmitted in an internet message.	"text/plain", "application/json", "image/jpeg", "application/pdf", "audio/mpeg", "video/mp4", "application/xml", "application/octet-stream", "application/vnd.ms-excel", "text/html"	VARCHAR2(100)	tech_info	-	-	-
22	sha1_d	Cryptographic hash function	"2fd4e1c67a2d28fced849ee1bb76e7391b93eb12"	CHAR(40)	tech_info, hash	REGEX - ^[0-9a-fA-F]{40}$	-	-
23	sha256_d	Cryptographic hash function	"5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9"	CHAR(64)	tech_info, hash	REGEX - ^[0-9a-fA-F]{64}$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
24	sha512_d	Cryptographic hash function	"cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327"	CHAR(128)	tech_info, hash	REGEX - ^[0-9a-fA-F]{128}$	-	-
25	short_name_d	Abbreviated or truncated form of a longer name or title.	"Ben", "Amy", "Max", "Mia", "Sam", "Zoe", "Leo", "Ava", "Eli", "Liv"	VARCHAR2(500)	-	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
26	`subnet_mask_d`	Numeric value used in combination with an IP address to define the network and host portions.	"255.255.255.0", "255.255.0.0", "255.255.255.128", "255.0.0.0", "255.255.255.252"	VARCHAR2(15)	tech_info	REGEX - ^(((255\.){3}(255|254|252|248|240|224|192|128|0))|((255\.){2}(255|254|252|248|240|224|192|128|0)\.0)|((255\.)(255|254|252|248|240|224|192|128|0)(\.0+){2})|((255|254|252|248|240|224|192|128|0)(\.0){3}))$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
27	`uri_d`	Uniform Resource Identifier, string of characters used to identify a resource.	"https://www.example.com", "ftp://ftp.example.org", "mailto:user@example.net", "tel:+1234567890", "file:///path/to/file.txt"	VARCHAR2(4000)	tech_info	-	-	-
28	`uuid4_d`	Universally Unique Identifier (UUID) version 4, a randomly generated identifier.	"550e8400-e29b-41d4-a716-446655440000", "123e4567-e89b-12d3-a456-426655440000", "a0a0a0a0-b0b0-cccc-1234-567890abcdef"	CHAR(36)	tech_info	REGEX - ^[0-9a-fA-F]{8}([-][0-9a-fA-F]{4}){3}[-][0-9a-fA-F]{12}$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-2 (Cont.) Tech Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
29	website_d	Collection of related web pages accessible through the World Wide Web.	"www.example.com", "blog.example.org", "shop.example.net", "www.google.com", "www.facebook.com", "www.twitter.com", "www.github.com", "www.wikipedia.org", "www.nytimes.com", "www.amazon.com"	VARCHAR2(4000))	tech_info	-	-	-

Table 10-3 Numeric Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
1	count_d	Refers to the numerical tally or total quantity of items, occurrences, or entities within a set or group. It represents the number of elements present and is used to quantify and track the abundance, frequency, or extent of something.	5, 12, 100, 1000	NUMBER	-	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-3 (Cont.) Numeric Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
2	`counter_d`	A metric type that represents a cumulative value that only ever increases over time. Counters are typically used to measure the total number of events or occurrences, such as the number of HTTP requests received, or the total bytes transferred.	1000, 50000, 8, 128, 56.4,	NUMBER	prometheus	-	-	-
3	`duration_d`	The length of time during which something continues or exists, measured from a starting point to an endpoint. Durations quantify the time elapsed between two events or within a specific period, providing a measure of time span, interval, or duration.	100, 1000, 250, 5000	NUMBER	-	-	-	-
4	`gauge_d`	Represents a single numerical value that can either increase or decrease over time, enabling monitoring of various metrics such as CPU usage, memory consumption, or network traffic.	1000, 50000, 8, 128, -22.78	NUMBER	prometheus	-	-	-
5	`histogram_count_d`	A metric associated with histograms. It represents the total count of observations or events recorded within the histogram's buckets over a specific time window. This count indicates how many times the observed metric falls within each bucket range, providing insight into the distribution of occurrences across different value ranges. This domain is typically used in conjunction with `histogram_sum_d`, `histogram_name_d`, and `histogram_upper_inclusive_bound_d` domains.	10, 100, 45, 123	NUMBER	prometheus, histogram	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-3 (Cont.) Numeric Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
6	`histogram_name_d`	A metric type used to measure the distribution of observations or events over a specified range of values. Each histogram has a name, which is a unique identifier used to reference and query the metric data. This domain is typically used in conjunction with `histogram_sum_d`, `histogram_count_d`, and `histogram_upper_inclusive_bound_d` domains.	"data_sent", "data_received", "request_timed_out"	VARCHAR2(1000)	prometheus, histogram	-	-	-
7	`histogram_sum_d`	A metric associated with histograms. It represents the sum of all observed values within the histogram's buckets over a specific time window. This sum can provide insights into the total cumulative value of the observed metric, such as the total duration of HTTP requests or the total amount of resource consumption within a given time period. This domain is typically used in conjunction with `histogram_name_d`, `histogram_count_d`, and `histogram_upper_inclusive_bound_d` domains.	10, 100, 45, 123	NUMBER	prometheus, histogram	-	-	-
8	`histogram_upper_inclusive_bound_d`	The upper limit of a bucket range, where values falling within this range are considered to be included in that bucket. This boundary indicates the maximum value that can be included in the bucket. For example, if a bucket has an upper inclusive bound of 100, it means that values up to and including 100 are counted within that bucket. This domain is typically in conjunction with `histogram_sum_d`, `histogram_count_d`, and `histogram_name_d domains`.	100, 500, 10, -20	NUMBER	prometheus, histogram	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-3 (Cont.) Numeric Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
9	`histogram_upper_inclusive_bound_string_d`	The textual representation of the upper inclusive bound of a bucket range. This string typically represents the maximum value that can be included in the bucket. For example, if a bucket has an upper inclusive bound string of "100", it means that values up to and including 100 are counted within that bucket.	"100", "500", "10", "-20"	VARCHAR2(1000)	prometheus, histogram	-	-	-
10	`mean_d`	The average value of a set of numbers, calculated by adding up all the values and dividing by the total count of numbers.	10, -2345, 4.67, 0	NUMBER	statistics	-	-	-
11	`median_d`	The middle value in a sorted list of numbers, separating the higher half from the lower half.	54, 20, 10, 1.45, -123	NUMBER	statistics	-	-	-
12	`mode_d`	The value that appears most frequently in a dataset.	4, 234, 100, -100, 0	NUMBER	statistics	-	-	-
13	`negative_number_d`	Denotes any numerical value less than zero. It represents quantities or values below the zero mark on the number line.	-1, -123, -1.456	NUMBER	-	CHECK <0	-	-
14	`non_negative_number_d`	Refers to any numerical value that is either zero or greater than zero. It includes zero and all positive numbers.	1, 123, 1.456, 0	NUMBER	-	CHECK >=0	-	-
15	`non_positive_number_d`	Refers to any numerical value that is either zero or less than zero. It includes both zero and negative numbers.	-1, -123, -1.456, 0	NUMBER	-	CHECK <=0	-	-
16	`non_zero_number_d`	Refers to any numerical value that is not equal to zero. It includes both positive and negative numbers, excluding zero	-1, -123, -1.456, 1, 123, 1.456	NUMBER	-	CHECK >0 or <0	-	-
17	`percent_change_d`	A measure that calculates the relative difference between two values expressed as a percentage, typically representing the change over a period of time. It represents a proportion out of 100 parts.	5, -5, 10, -10, -66.4	NUMBER	-	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-3 (Cont.) Numeric Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
18	`percent_d`	A unit of measurement denoting a portion or fraction of a whole, expressed as a fraction of 100. It represents a proportion out of 100 parts and is commonly used to indicate relative quantities, rates, or comparisons, such as percentages of increase, decrease, or distribution. Percentages can be negative.	100, 98, 0.50, 75, -300	NUMBER	-	-	-	-
19	`percentile_d`	A statistical measure that indicates the percentage of data points in a distribution that are equal to or below a given value.	0, 10, 50 , 99, 33.33	NUMBER	-	CHECK >=0	-	-
20	`positive_number_d`	Refers to any numerical value greater than zero. It represents quantities, measurements, or values that are above the zero mark on the number line.	1, 123, 1.456	NUMBER	-	CHECK >0	-	-
21	`ratio_d`	A comparison of the magnitude of two quantities, often expressed as the quotient of one divided by the other.	0.5, 0.33, 1.5, 1	NUMBER	-	-	-	-
22	`standard_deviation_d`	The measure of the amount of variation or dispersion in a set of data.	0, 1, 10, 34, 56.56	NUMBER	statistics	CHECK >= 0	-	-
23	`unit_count_d`	The numerical quantity or tally representing the number of individual units of a particular item or entity. It indicates the quantity of units present and is used to measure the abundance or quantity of items within a set or group. This domain is typically used in conjunction with `unit_id_d` and `unit_price_d` that represent the ID and price (the monetary value in an arbitrary currency) of a unit respectively.	5, 10, 1, 0, 1000	NUMBER	unit	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-3 (Cont.) Numeric Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
24	`unit_id_d`	A unique identifier or code assigned to a specific unit of a product, item, or entity. It distinguishes one unit from another and is used for tracking, inventory management, and reference purposes. This domain is typically used in conjunction with `unit_count_d` and `unit_price_d` that represent the count and price (the monetary value in an arbitrary currency) of a unit respectively.	50061, 50062, 123, 100001	NUMBER	unit	-	-	-
25	`unit_price_d`	The monetary value or cost associated with a single unit of a product, item, or entity. It represents the price charged or paid for each unit, and is used to calculate the total cost or revenue generated from the sale or purchase of units. This domain is typically used in conjunction with `unit_count_d` and `unit_id_d` that represent the count and ID of a unit respectively.	1, 15, 3.99, 500.67	NUMBER	unit	-	-	-
26	`variance_d`	The measure of the dispersion or spread of a set of data points around their mean.	0, 1, 10, 34, 56.56	NUMBER	statistics	CHECK >= 0	-	-

Table 10-4 Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
1	`binary_d`	Base-2 numeral system representing numbers using only two symbols, typically 0 and 1. Each digit represents a power of 2. For example, the binary number 1010 represents the decimal number 10.	"01010001111", "0", "1", "00001111"	VARCHAR2(4000)	-	REGEX - ^[01]*$	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
2	`currency_amount_d`	Refers to the numerical value assigned to a specific currency unit in a financial transaction or monetary exchange. It represents the quantity or amount of currency being transacted. This domain is typically used in conjunction with the `currency_code_d` domain that represents the currency in which the transaction is done.	10, 1000, 5000.67, 100000	NUMBER	currency	-	-	-
3	`currency_code_d`	Refers to a code, generally three lettered, that represents a specific currency in international transactions, such as USD for United States Dollar or EUR for Euro. This domain is typically used in conjunction with the `currency_amount_d` domain that represents the numerical value of the monetary transaction.	"USD", "INR", "EUR", "GBP"	VARCHAR2(3)	currency	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
4	day_d	The name of the day of the week.	"SUNDAY", "MONDAY", ..., "SATURDAY"	CHAR(9)	-	List - ["SUNDAY", "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "SATURDAY"]	Based on NLS Property 1. SUNDAY 2. MONDAY 3. TUEDAY 4. WEDDAY 5. THUDAY 6. FRIDAY 7. SATDAY OR 1. MONDAY 2. TUEDAY 3. WED	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
							DAY 4. THUDAY 5. FRIDAY 6. SATDAY 7. SUNDAY	

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
5	day_short_d	Abbreviated representation of the name of days of the week.	"SUN","MON", ..., "SAT"	CHAR(3)	-	List - ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"]	Based on NLS Property 1. SUN 2. MON 3. TUE 4. WED 5. THU 6. FRI 7. SAT OR 1. MON 2. TUE 3. WED 4. THU 5. FRI 6. SAT 7. SUN	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
6	day_enum_d	Refers to an enumeration representing the seven days of the week. The domain supports long day names (MONDAY) and short day names (MON)	day_enum_d.MONDAY,day_enum_d.MON,day_enum_d.TUESDAY,day_enum_d.TUE	ENUM	calendar_info	ENUM - day_enum_d.MONDAY,day_enum_d.MON,day_enum_d.TUESDAY,day_enum_d.TUE	-	-
7	hexadecimal_d	Base-16 numeral system utilizing 16 symbols, including 0-9 and A-F to represent values from 10 to 15. Each digit represents a power of 16. For example, the hexadecimal number A8 represents the decimal number 168.	"001122AAFFF", "fffff", "00ffaa"	VARCHAR2(4000)	-	REGEX - ^[0-9a-fA-F]*$	-	-
8	latitude_d	Geographical coordinate that specifies the north-south position of a point on the Earth's surface relative to the equator. It is measured in degrees, ranging from 90 degrees north (at the North Pole) to 90 degrees south (at the South Pole).	37.773972	NUMBER	latlong, geographic	CHECK -90<=latitude<=90	-	-
9	longitude_d	Geographical coordinate that indicates the east-west position of a point on the Earth's surface relative to the Prime Meridian. It is measured in degrees, ranging from 180 degrees west to 180 degrees east.	-122.431297	NUMBER	latlong, geographic	CHECK -180<=longitude<=180	-	-
10	measure_d	Refers to the quantification or assessment of a particular attribute, characteristic, or quantity of an object, phenomenon, or system.	500, 1000, 1500, -22.3	NUMBER	-	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
11	`measure_type_d`	This refers to the category or classification of a specific measurement. It defines the nature or type of the measurement being taken, such as weight, temperature, time, or quantity. This domain is used in conjunction with the `measure_value_d` domain that represents the numerical value of the measure.	“kg/cm^3”, “m”, “meters”, “celsius”	VARCHAR2(100)	measure	-	-	-
12	`measure_value_d`	The numerical or quantitative representation of a measurement within a given measure type. It indicates the magnitude or amount of the observed phenomenon. This domain is used in conjunction with the `measure_type_d` domain that represents the unit of the measure.	1000, 45.67, -300, 0	NUMBER	measure	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
13	`month_d`	The name of the months.	"JANUARY", "FEBRUARY", ..., "DECEMBER"	CHAR(10)	-	List - ["JANUARY", "FEBRUARY", ..., "DECEMBER"]	1. JANUARY 2. FEBRUARY 3. MARCH 4. APRIL 5. MAY 6. JUNE 7. JULY 8. AUGUST 9. SEPTEMBER 10. OCTOBER 11. NOVEM	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
							BER 12. DECEMBER	
14	month_short_d	The abbreviated representation of the name of months in a year.	"JAN", "FEB", ..., "DEC"	CHAR(3)	-	List - ["JAN", "FEB", ..., "DEC"]	1. JAN 2. FEB 3. MAR 4. APR 5. MAY 6. JUN 7. JUL 8. AUG 9. SEP 10. OCT 11. NOV 12. DEC	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
15	month_enum_d	Refers to an enumeration that represents the twelve months of the year. The domain supports long month names (JANUARY) and short month names (JAN).	month_enum_d.JANUARY, month_enum_d.JAN, month_enum_d.FEBRUARY,month_enum_d.FEB	ENUM	calendar_info	ENUM - month_enum_d.JANUARY, month_enum_d.JAN, month_enum_d.FEBRUARY,month_enum_d.FEB, and so on.	-	-
16	octal_d	Base-8 numeral system utilizing eight symbols, usually 0 to 7. Each digit represents a power of 8. For instance, the octal number 12 represents the decimal number 10.	"12345", "7777", "01010"	VARCHAR2(4000)	-	REGEX - ^[0-7]*$	-	-
17	rank_d	The position given to something compared to others based on a particular standard, indicating its place in order. Rank is usually a positive integer.	1, 2,3, 10, 100, 5000	INTEGER	assessment	CHECK >= 1	-	-
18	rating_d	Denotes a numerical or qualitative assessment given to evaluate the quality, value, or performance of something based on predefined criteria. Ratings help users make informed decisions by providing an indication of satisfaction, effectiveness, or relevance.	-100, 0, 1, 45, 300	NUMBER	assessment	-	-	-
19	score_d	Represents a numerical evaluation or rating assigned to measure performance, quality, or success in various contexts. Scores provide a standardized way to assess and compare performance levels or outcomes across different situations.	-100, 0, 1, 45, 300	NUMBER	assessment	-	-	-

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 10-4 (Cont.) Miscellaneous Built-in Domains

#	Domain Name	Description	Example	Data Type	Annotation	Constraint	Domain Order	Domain Display
20	`time_zone_abbreviation_d`	Shortened form or acronym representing the name of a specific time zone. It is typically composed of letters and is used to identify a region's time standard quickly. Examples include "EST" for Eastern Standard Time or "CET" for Central European Time. This domain can be used in the place of `time_zone_utc_offset_d`, which represents the timezone in numerical form.	"PST", "IST", "EST"	VARCHAR2(3)	timezone	-	-	-
21	`time_zone_utc_offset_d`	The numerical difference in hours and minutes between a specific time zone's local time and Coordinated Universal Time (UTC). It represents the deviation from the standard UTC time, which serves as a reference point for coordinating time globally. Positive offsets indicate locations ahead of UTC, while negative offsets indicate locations behind UTC. This domain can be used in the place of `time_zone_abbreviation_d`, which represents the acronym of the timezone.	"UTC+04:00", "UTC-03:30"	VARCHAR2(20)	timezone	-	-	-
22	`year_d`	The year in numeric form. The year 1 BCE is numbered 0, the year 2 BCE is numbered −1, and in general the year n BCE is numbered "−(n − 1)" (a negative number equal to 1 − n). This system of numbering is often referred to as "Astronomical Year Numbering."	-22, 1996, 2000, 1800	NUMBER(4)	-	-	-	-

10.2 Schema Annotations

This section explains how you can use schema annotations (hereinafter "annotations") for database objects.

For many applications, it is important to maintain additional property metadata for database objects such as tables, views, table columns, indexes, and domains. Annotations enable your applications to store and retrieve additional user-specific metadata about database objects and

Copyright © 1999, 2025, Oracle and/or its affiliates.

table columns. Applications can use such metadata to help render effective user interfaces and customize application logic.

Topics:

- Overview of Annotations
- Annotations and Comments
- Supported Database Objects
- Privileges Required for Using Annotations
- DDL Statements for Annotations

10.2.1 Overview of Annotations

What are annotations and where can you use them?

Annotations are a lightweight declarative facility for developers to centrally register usage properties for database schema objects. Annotations are stored in dictionary tables and are available to any application looking to standardize behavior across common data in related applications. Annotations are not interpreted by the database in any way. They are custom data properties for database metadata - included for table columns, tables, and indexes, that applications can use as additional property metadata to render user interfaces or customize application logic.

Being a mechanism to define and store application metadata centrally in the database, annotations enable you to share the metadata information across applications, modules and microservices. You can add annotations to schema objects when you create new objects (using the `CREATE` statements) or modify existing objects (using the `ALTER` statements).

Annotating the data model with metadata provides additional data integrity, consistency and data model documentation benefits. Your applications can store user-defined metadata for database objects and table columns that other applications or users can retrieve and use. Storing the metadata along with the data guarantees consistency and universal accessibility to any user or application that uses the data.

An individual annotation has a name and an optional value. The name and the optional value are freeform text fields. For example, you can have an annotation with a name and value pair, such as `Display_Label 'Employee Salary'`, or you can have a standalone annotation with only a name, such as `UI_Hidden`, which does not need a value because the name is self-explanatory.

The following are further details about annotations.

- When an annotation name is specified for a schema object using the `CREATE` DDL statement, an annotation is automatically created.
- Annotations are additive, meaning that you can specify multiple annotations for the same schema object.
- You can add multiple annotations at once to a schema object using a single DDL statement. Similarly, a single DDL statement can drop multiple annotations from a schema object.
- An annotation is represented as a subordinate element to the database object to which the annotation is added. Annotations do not create a new object type inside the database.
- You can add annotations to any schema object that supports annotations provided you own or have alter privileges on the schema object. You do not need to schema qualify the annotation name.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- You can issue SQL queries on dictionary views to obtain all annotations, including their names and values, and their usage with schema objects.

10.2.2 Annotations and Comments

You can also annotate database objects such as tables and table columns using the `COMMENT` command. Comments that are associated with schema objects are stored in data dictionaries along with the metadata of the objects.

Annotations are simpler to use and have a broader scope than comments. Following are the major differences between comments and annotations:

- Comments are a commenting mechanism used to add metadata to only certain schema objects, such as tables and columns. Comments are not available for other schema objects, such as indexes, procedures, triggers, and domains.
- Comments do not have a name, they have only a freeform value.
- Comments are not additive meaning that you cannot add multiple comments for the same object. Specifying a new comment overwrites the prior comment for the corresponding table or column.
- You need separate DDL statements for comments whereas you can combine multiple annotations into one DDL statement.
- A separate set of dictionary views exists for different entities. For instance, there is a view for table comments and another view for column comments. Annotations, on the other hand, are unified across all object types, which makes them simpler to query and use.

> **See Also**
>
> Comments for more information about using `COMMENTS`.

10.2.3 Supported Database Objects

Annotations are supported for the following database objects.

- Tables and table columns
- Views and view columns
- Materialized views and materialized view columns
- Indexes
- Domains and multi-column domain columns

10.2.4 Privileges Required for Using Annotations

To add or drop annotations, you require the `CREATE` or `ALTER` privilege on the schema object for which the annotation is specified in the `CREATE` or `ALTER` DDL statements.

You cannot create or remove annotations explicitly. Annotations are automatically created when they are used for the first time. Since annotations are stored as subordinate elements within the database object on which they are defined, adding annotations does not create a new object type inside the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.2.5 DDL Statements for Annotations

This section explains the annotation syntax and provides the DDL statements to define or alter annotations for tables, table columns, views, materialized views, indexes, and domains.

Topics:

- Annotation Syntax
- DDL Statements to Annotate a Table
- DDL Statements to Annotate a Table Column
- DDL Statements to Annotate Views and Materialized Views
- DDL Statements to Annotate Indexes
- DDL Statements to Annotate Domains
- Dictionary Table and Views

10.2.5.1 Annotation Syntax

The following snippet illustrates the annotation syntax that is used in DDL statements to define annotations for tables, table columns, views, materialized views, indexes, and domains.

```
annotations
         ::= 'ANNOTATIONS' '(' annotations_list ')'

annotations_list
         ::= ( 'ADD' ('IF NOT EXISTS' | 'OR REPLACE' )? | 'DROP' 'IF EXISTS'? | REPLACE)?
 annotation ( ',' ( 'ADD' ('IF NOT EXISTS' | 'OR REPLACE' )? | 'DROP' 'IF EXISTS'? |
REPLACE)?
 annotation )*

annotation
         ::= annotation_name annotation_value?

annotation_name
         ::= identifier

annotation_value
         ::= character_string_literal
```

<annotation_value> is a character string literal that can hold up to 4000 characters.

<annotation_name> is an identifier and has the following requirements:

- An identifier can have up to 1024 characters.
- If an identifier is a reserved word, you must provide the annotation name in double quotes.
- An identifier with double quotes can contain white characters.
- An identifier that contains only white characters is not allowed.

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.2.5.2 DDL Statements to Annotate a Table

You can use the following DDL statements to annotate a table when creating or altering the table.

Using the `CREATE TABLE` Statement

The following are examples of the `CREATE TABLE` statement with annotations.

The following example adds an annotation: `Operation` with a JSON value, and another annotation: `Hidden`, which is standalone and has no value.

```
CREATE TABLE Table1 (
  T NUMBER)
  ANNOTATIONS(Operations '["Sort", "Group"]', Hidden);
```

Adding an annotation can be preceded by the `ADD` keyword. The `ADD` keyword is considered to be the default operation, if nothing is specified.

The following example uses the optional `ADD` keyword to add the `Hidden` annotation (which is also standalone) to `Table2`.

```
CREATE TABLE Table2 (
  T NUMBER)
  ANNOTATIONS (ADD Hidden);
```

The `ADD` keyword is an implicit operation when annotations are defined, and can be omitted.

Using the `ALTER TABLE` Statement

In the following example, the `ALTER TABLE` command drops all annotation values for the following annotation names: `Operations` and `Hidden`.

```
ALTER TABLE Table1
  ANNOTATIONS(DROP Operations, DROP Hidden);
```

The following example has the `ALTER TABLE` command to add `JoinOperations` annotation with `Join` value, and to drop the annotation name: `Hidden`. When dropping an annotation, you need to only include the annotation name with the `DROP` command.

```
ALTER TABLE Table1
  ANNOTATIONS(ADD JoinOperations 'Join', DROP Hidden);
```

Due to the Hidden annotation being dropped in the earlier statement, the output is: `ORA-11553: Annotation name 'HIDDEN' does not exist for the object 'Table1'`. To avoid this error: when the annotation does not exist, you can use the IF EXISTS clause.

```
ALTER TABLE Table1 ANNOTATIONS(ADD JoinOperations 'Join', DROP IF EXISTS Hidden);
```

Multiple `ADD` and `DROP` keywords can be specified in one DDL statement.

Trying to re-add an annotation with a different value for the same object (for object-level annotations) or for the same column (for column-level annotations) raises an error. For instance, the following statement fails:

```
ALTER TABLE Table1
  ANNOTATIONS(ADD JoinOperations 'Join Ops');
```

The output is:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
ORA-11552: Annotation name 'JOINOPERATIONS' already exists for the object 'TABLE1'.
```

As an alternative, the annotation syntax allows you to replace an annotation value using the `REPLACE` keyword. The following statement replaces the value of `JoinOperations` with '`Join Ops`':

```
ALTER TABLE Table1
  ANNOTATIONS(REPLACE JoinOperations 'Join Ops');
```

The output is

```
Table altered.
```

Alternatively, to avoid an error when an annotation already exists, you can use the `IF NOT EXISTS` clause. The following statement adds the `JoinOperations` annotation only if it does not exist. If the annotation exists, the annotation value is unchanged and no error is raised.

```
ALTER TABLE Table1
  ANNOTATIONS(ADD IF NOT EXISTS JoinOperations 'Join Ops');
```

Similarly, dropping a non-existent annotation raises an error:

```
ALTER TABLE Table1 ANNOTATIONS(DROP Title);
```

The output is:

```
ORA-11553: Annotation name 'TITLE' does not exist.
```

To avoid the error, the `IF EXISTS` clause can be used, as follows:

```
ALTER TABLE Table1
  ANNOTATIONS(DROP IF EXISTS Title);
```

The output is:

```
Table altered.
```

> **See Also**
>
> CREATE TABLE and ALTER TABLE in SQL Language Reference for complete clause changes and definitions.

10.2.5.3 DDL Statements to Annotate a Table Column

You can use the following DDL statements to annotate a table column when creating or altering the table.

Using the `CREATE TABLE` Statement

To add column-level annotations using a `CREATE TABLE` statement, specify the annotations as a part of the `column_definition` clause. Annotations are specified at the end, after inline constraints.

The following examples specify annotations for columns at table creation.

```
CREATE TABLE Table1
  (T NUMBER ANNOTATIONS(Operations 'Sort', Hidden));

CREATE TABLE Table2
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  (T NUMBER ANNOTATIONS (Hidden));
```

The following example specifies table-level and column-level annotations for the `Employee` table.

```
CREATE TABLE Employee (
  Id NUMBER(5) ANNOTATIONS(Identity, Display 'Employee ID', "Group" 'Emp_Info'),
  Ename VARCHAR2(50) ANNOTATIONS(Display 'Employee Name', "Group" 'Emp_Info'),
  Sal NUMBER ANNOTATIONS(Display 'Employee Salary', UI_Hidden)
)
 ANNOTATIONS (Display 'Employee Table');
```

The `Employee` table in the previous example has column-level and object-level annotations with `Display` annotations defined at the column level and object level. You can define a new annotation with the same name as long as it corresponds to a different object, column pair. For instance, you cannot define another annotation with the `Display` name for the `Employee` table but you can define new annotations with `Display` name for every column of the `Employee` table. You can also define other annotations such as "`Group`" for the `Sal` column. The annotation "`Group`" needs to be double quoted because it is a reserved word.

Using the `ALTER TABLE` Statement

To add column-level annotations using an `ALTER TABLE` statement, specify the annotations as a part of the `modify_col_properties` clause. Annotations are specified at the end, after inline constraints.

The following example adds a new `Identity` annotation for the column T of `Table1` table.

```
ALTER TABLE Table1
  MODIFY T ANNOTATIONS(Identity 'ID');
```

The following example adds a `Label` annotation and drops the `Identity` annotation.

```
ALTER TABLE Table1
  MODIFY T ANNOTATIONS(ADD Label, DROP Identity);
```

> **See Also**
>
> CREATE TABLE and ALTER TABLE in SQL Language Reference for complete clause changes and definitions.

10.2.5.4 DDL Statements to Annotate Views and Materialized Views

You can use the following DDL statements to annotate a view and a materialized view.

Views and materialized views support annotations at the view level and column level. Column-level annotations are only supported for table views as a part of the column alias definition clause.

Using the `CREATE VIEW` Statement

The following example shows the view-level and column-level annotations:

```
CREATE OR REPLACE VIEW HighWageEmp
(
  Id ANNOTATIONS(Identity, Display 'Employee ID', "Group" 'Emp_Info'),
  Ename ANNOTATIONS (Display 'Employee Name',  "Group" 'Emp_Info'),
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  Sal  ANNOTATIONS (Display 'Emp Salary')
)
ANNOTATIONS (Title 'High Wage Employee View')
AS SELECT * FROM EMPLOYEE WHERE Sal > 100000;
```

Using the `CREATE MATERIALIZED VIEW` Statement

The following example adds annotation at the view level in the `Materialized View` statement:

```
CREATE MATERIALIZED VIEW MView1
  ANNOTATIONS (Title 'Tab1 MV1', ADD Snapshot)
  AS SELECT * FROM Table1;
```

The following example adds annotation at the view level and column level in the `Materialized View` statement:

```
CREATE MATERIALIZED VIEW MView1(
  T ANNOTATIONS (Hidden))
 ANNOTATIONS (Title 'Tab1 MV1', ADD Snapshot)
 AS SELECT * FROM Table1;
```

Using the `ALTER VIEW` Statement

To provide support for annotations, the `ALTER VIEW` statement has a new sub-clause added to it that allows altering annotations at the view level. Other sub-clauses that are supported in the `ALTER VIEW` statement are: modifying view constraints, enabling recompilation, and changing `EDITIONABLE` property.

Column-level annotations for views cannot be altered. Previously added column-level annotations can be dropped only by dropping the view and recreating a new one.

The following example is an `ALTER VIEW` statement that drops the `Title` annotation and adds the `Identity` annotation at the view level.

```
ALTER VIEW HighWageEmp
  ANNOTATIONS(DROP Title, ADD Identity);
```

Using the `ALTER MATERIALIZED VIEW` Statement

The `ALTER MATERIALIZED VIEW` statement has a new sub-clause that is added to alter annotations globally at the materialized view level. Column-level annotations can be dropped only by dropping and recreating the materialized view.

The following `ALTER MATERIALIZED VIEW` statement drops `Snapshot` annotation from `MView1`.

```
ALTER MATERIALIZED VIEW MView1
  ANNOTATIONS(DROP Snapshot);
```

> **See Also**
>
> CREATE VIEW, ALTER VIEW, CREATE MATERIALIZED VIEW, and ALTER MATERIALIZED VIEW in SQL Language Reference for complete clause changes and definitions.

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.2.5.5 DDL Statements to Annotate Indexes

You can use the following DDL statements to annotate an index.

Using the `CREATE INDEX` Statement

The following are examples of creating index-level annotations.

```
CREATE TABLE DEPARTMENT(
  DEPT_ID NUMBER,
  UNIT NUMBER);

CREATE INDEX I_DEPT_ID ON DEPARTMENT(DEPT_ID)
  ANNOTATIONS(Display 'Department Index');

CREATE UNIQUE INDEX UI_UNIT ON DEPARTMENT(UNIT)
  ANNOTATIONS(Display 'Department Unique Index');
```

Using the `ALTER INDEX` Statement

The following is an example of altering an index-level annotation.

```
ALTER INDEX I_DEPT_ID
  ANNOTATIONS(DROP Display, ADD ColumnGroup 'Group1');
```

> **See Also**
>
> CREATE INDEX and ALTER INDEX in SQL Language Reference for complete clause changes and definitions.

10.2.5.6 DDL Statements to Annotate Domains

You can use the following DDL statements to annotate a domain.

You can specify annotations for domains at the domain level or at the column level. Annotations defined on domains are inherited to objects that reference the domain. Annotations are allowed only for regular domains and not for flexible domains.

To specify domain-level annotations for single-column domains, the syntax with parentheses (used for multi-column domains) is required to distinguish between column-level and object-level annotations. Otherwise, the annotations for single column domains are considered column level.

Using the `CREATE DOMAIN` Statement

The following example creates domain annotations by specifying column-level annotations for a single-column domain.

```
CREATE DOMAIN dept_codes_1 AS NUMBER(3)
  CONSTRAINT dept_chk_1 CHECK (dept_codes_1 > 99 AND dept_codes_1 != 200)
  ANNOTATIONS (Title 'Column level annotation');
```

The following examples specify a domain-level annotation for a single-column domain. This requires the use of the domains syntax for multi-column domains (with parentheses for columns).

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CREATE DOMAIN dept_codes_2 AS (
  code AS NUMBER(3)
  CONSTRAINT dept_chk_2 CHECK (code > 99 AND code != 200))
  ANNOTATIONS (Title 'Domain Annotation');

CREATE DOMAIN HourlyWages AS Number(10)
  DEFAULT ON NULL 15
  CONSTRAINT MinimalWage CHECK (HourlyWages > = 7 and HourlyWages <=1000) ENABLE
  DISPLAY TO_CHAR(HourlyWages, '$999.99')
  ORDER ( -1*HourlyWages )
  ANNOTATIONS (Title 'Column level annotation');
```

The following example creates a multi-column domain with annotations at the column and domain levels.

```
CREATE DOMAIN US_City AS (
    name  AS VARCHAR2(50) ANNOTATIONS (Address),
    state AS VARCHAR2(50) ANNOTATIONS (Address),
    zip AS NUMBER ANNOTATIONS (Address)
  )
  CONSTRAINT City_CK CHECK(state in ('CA','AZ','TX') and zip < 100000)
  DISPLAY name || ', ' || state || ' , ' || TO_CHAR(zip)
  ORDER state || ', ' || TO_CHAR(zip) || ', ' || name
  ANNOTATIONS (Title 'Domain Annotation');
```

Using the `ALTER DOMAIN` Statement

Object-level annotations can be modified with `ALTER` statements. Column-level annotations for domains cannot be altered. Previously added column-level annotations can be dropped only by dropping the domain and recreating a new one.

The following example alters domain-level annotation for the `dept_codes_2` domain.

```
ALTER DOMAIN dept_codes_2
  ANNOTATIONS(DROP Title, ADD Name 'Domain');
```

> **See Also**
>
> CREATE DOMAIN and ALTER DOMAIN in SQL Language Reference for complete clause changes and definitions.

10.2.5.7 Dictionary Table and Views

A dictionary table called `ANNOTATIONS_USAGE$` includes all the usage of annotations with schema objects, such as tables and views. When a new annotation name, value, or both are specified to a schema object, a new entry in `ANNOTATIONS_USAGE$` table is created. Similarly, when a schema object drops an annotation, the corresponding entry is dropped from the `ANNOTATIONS_USAGE$` table.

The following dictionary views track the list of annotations and their usage across all schema objects:

- `{DBA|USER|ALL|CDB}_ANNOTATIONS`
- `{DBA|USER|ALL|CDB}_ANNOTATIONS_USAGE`
- `{DBA|USER|ALL|CDB}_ANNOTATIONS_VALUES`

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.2.5.7.1 Querying Dictionary Views

You can run queries on dictionary views to obtain the list of annotations that are used for specific schema objects.

Here are examples of query statements issued on an 'EMP' table:

To obtain table-level annotations for the ‘EMP’ table:

```
SELECT * FROM USER_ANNOTATIONS_USAGE
  WHERE Object_Name = 'EMPLOYEE'
   AND Object_Type = 'TABLE'
   AND Column_Name IS NULL;
```

To obtain column-level annotations for the ‘EMP’ table:

```
SELECT * FROM USER_ANNOTATIONS_USAGE
  WHERE Object_Name = 'EMPLOYEE'
   AND Object_Type = 'TABLE'
   AND Column_Name IS NOT NULL;
```

To obtain column-level annotations for the ‘EMP’ table as a single JSON collection per column:

```
SELECT  U.Column_Name, JSON_ARRAYAGG(JSON_OBJECT(U.Annotation_Name, U.Annotation_Value))
  FROM USER_ANNOTATIONS_USAGE U
  WHERE Object_Name = 'EMPLOYEE' AND Object_Type = 'TABLE' AND Column_Name IS NOT NULL
  GROUP BY Column_Name;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

11

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

11.1 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:`

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `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.

11.2 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 11-1. 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 11-2. The functions have additional options (for example, the character position at which to start searching the string for the pattern).

Table 11-1 Oracle SQL Pattern-Matching Condition and Functions

Name	Description
`REGEXP_LIKE`	**Condition** that can appear in the `WHERE` clause of a query, causing the query to return rows that match the given pattern. **Example:** This `WHERE` clause identifies employees with the first name of Steven or Stephen: `WHERE REGEXP_LIKE((hr.employees.first_name, '^Ste(v|ph)en$')`
`REGEXP_COUNT`	**Function** that returns the number of times the given pattern appears in the given string. **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: `REGEXP_COUNT('Albert Einstein', 'e', 7, 'c')` (The returned value is 1, because the `c` option specifies case-sensitive matching.)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-1 (Cont.) Oracle SQL Pattern-Matching Condition and Functions

Name	Description
REGEXP_INSTR	**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. **Example:** This function invocation returns the starting position of the first valid email address in the column `hr.employees.email`: `REGEXP_INSTR(hr.employees.email, '\w+@\w+(\.\w+)+')` If the returned value is greater than zero, then the column contains a valid email address.
REGEXP_REPLACE	**Function** that returns the string that results from replacing occurrences of the given pattern in the given string with a replacement string. **Example:** This function invocation puts a space after each character in the column `hr.countries.country_name`: `REGEXP_REPLACE(hr.countries.country_name, '(.)', '\1 ')`
REGEXP_SUBSTR	**Function** that is like `REGEXP_INSTR` except that instead of returning the starting position of the given pattern in the given string, it returns the matching substring itself. **Example:** This function invocation returns `'Oracle'` because the `x` option ignores the spaces in the pattern: `REGEXP_SUBSTR('Oracle 2010', 'O r a c l e', 1, 1, 'x')`

Table 11-2 describes the pattern-matching options that are available to each pattern matcher in Table 11-1.

Table 11-2 Oracle SQL Pattern-Matching Options for Condition and Functions

Pattern-Matching Option	Description	Example
`i`	Specifies case-insensitive matching.	This function invocation returns 3: `REGEXP_COUNT('Albert Einstein', 'e', '`**`i`**`')`
`c`	Specifies case-sensitive matching.	This function invocation returns 2: `REGEXP_COUNT('Albert Einstein', 'e', '`**`c`**`')`
`n`	Allows the Dot operator (`.`) to match the newline character, which is not the default (see Table 11-3).	In this function invocation, the string and search pattern match only because the `n` option is specified: `REGEXP_SUBSTR('a'||CHR(10)||'d', 'a.d', 1, 1, '`**`n`**`')`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-2 (Cont.) Oracle SQL Pattern-Matching Options for Condition and Functions

Pattern-Matching Option	Description	Example
`m`	Specifies **multiline mode**, where a newline character inside a string terminates a line. The string can contain multiple lines. Multiline mode affects POSIX operators Beginning-of-Line Anchor (^) and End-of-Line Anchor ($) (described in Table 11-3) but not PERL-influenced operators \A, \Z, and \z (described in Table 11-5).	This function invocation returns `ac`: `REGEXP_SUBSTR('ab'||CHR(10)||'ac', '^a.', 1, 2, 'm')`
`x`	Ignores whitespace characters in the search pattern. By default, whitespace characters match themselves.	This function invocation returns `abcd`: `REGEXP_SUBSTR('abcd', 'a b c d', 1, 1, 'x')`

See Also

Oracle Database SQL Language Reference fore more information about single row functions

11.3 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.
- 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
- 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 (?))

See Also

- POSIX Operators in Oracle SQL Regular Expressions
- POSIX Regular Expressions Specification
- Oracle SQL Multilingual Extensions to POSIX Standard
- Oracle SQL PERL-Influenced Extensions to POSIX Standard

Copyright © 1999, 2025, Oracle and/or its affiliates.

11.4 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

11.4.1 POSIX Operators in Oracle SQL Regular Expressions

Table 11-3 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 11-3 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 n (see Table 11-2). The Linux, UNIX, and Windows platforms recognize the newline character as the linefeed character (`\x0a`). The Macintosh platforms recognize the newline character as the carriage return character (`\x0d`). **Note:** In the POSIX standard, this operator matches any English character except NULL and the newline character.	The expression **`a.b`** matches the strings `abb`, `acb`, and `adb`, but does not match `acc`.
+	One or More Plus Quantifier	Matches one or more occurrences of the preceding subexpression (greedy[1]).	The expression **`a+`** matches the strings `a`, `aa`, and `aaa`, but does not match `ba` or `ab`.
*	Zero or More Star Quantifier	Matches zero or more occurrences of the preceding subexpression (greedy[1]).	The expression **`ab*c`** matches the strings `ac`, `abc`, and `abbc`, but does not match `abb` or `bbc`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-3 (Cont.) POSIX Operators in Oracle SQL Regular Expressions

Operator Syntax	Names	Description	Examples
?	Zero or One Question Mark Quantifier	Matches zero or one occurrences of the preceding subexpression (greedy[1]).	The expression **ab?c** matches the strings `abc` and `ac`, but does not match `abbc` or `adc`.
{*m*}	Interval Exact Count	Matches exactly *m* occurrences of the preceding subexpression.	The expression **a{3}** matches the string `aaa`, but does not match `aa`.
{*m*,}	Interval At-Least Count	Matches at least *m* occurrences of the preceding subexpression (greedy[1]).	The expression **a{3,}** matches the strings `aaa` and `aaaa`, but does not match `aa`.
{*m*,*n*}	Interval Between Count	Matches at least *m* but not more than *n* occurrences of the preceding subexpression (greedy[1]).	The expression **a{3,5}** matches the strings `aaa`, `aaaa`, and `aaaaa`, but does not match `aa` or `aaaaaa`.
[*char*...]	Matching Character List	Matches any single character in the list within the brackets. In the list, all operators except these are treated as literals: • Range operator: `-` • POSIX character class: `[: :]` • POSIX collation element: `[. .]` • POSIX character equivalence class: `[= =]` A dash (`-`) is a literal when it occurs first or last in the list, or as an ending range point in a range expression, as in `[#--]`. A right bracket (`]`) is treated as a literal if it occurs first in the list. **Note:** In the POSIX standard, a range includes all collation elements between the start and end of the range in the linguistic definition of the current locale. Thus, ranges are linguistic rather than byte value ranges; the semantics of the range expression are independent of the character set. In Oracle Database, the linguistic range is determined by the `NLS_SORT` initialization parameter.	The expression **[abc]** matches the first character in the strings `all`, `bill`, and `cold`, but does not match any characters in `doll`.
[^*char*...]	Nonmatching Character List	Matches any single character *not* 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 **[^abc]def** matches the string `xdef`, but not `adef`, `bdef`, or `cdef`. The expression **[^a-i]x** matches the string `jx`, but does not match `ax`, `fx`, or `ix`.
[*alt1*|*alt2*]	Or	Matches either alternative.	The expression **a|b** matches the character `a` or `b`.
(*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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-3 (Cont.) POSIX Operators in Oracle SQL Regular Expressions

Operator Syntax	Names	Description	Examples
\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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-3 (Cont.) POSIX Operators in Oracle SQL Regular Expressions

Operator Syntax	Names	Description	Examples
`[=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 *Oracle Database Globalization Support Guide*.	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 11-5).

2 Specify multiline mode with the pattern-matching option `m`, described in Table 11-2.

11.4.2 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 11-4 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 11-4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-4 (Cont.) POSIX Operators and Multilingual Operator Relationships

Operator	POSIX BRE Syntax	POSIX ERE Syntax	Multilingual Enhancement
[==]	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.

11.4.3 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 11-5 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 11-5 PERL-Influenced Operators in Oracle SQL Regular Expressions

Operator Syntax	Description	Examples
\d	Matches a digit character. Equivalent to POSIX expression [[:digit:]].	The expression **^\(\d{3}\) \d{3}-\d{4}$** matches (650) 555-0100 but does not match 650-555-0100.
\D	Matches a nondigit character. Equivalent to POSIX expression [^[:digit:]].	The expression **\w\d\D** matches b2b and b2_ but does not match b22.
\w	Matches a word character (that is, an alphanumeric or underscore (_) character). Equivalent to POSIX expression [[:alnum:]_].	The expression **\w+@\w+(\.\w+)+** matches the string jdoe@company.co.uk but does not match jdoe@company.
\W	Matches a nonword character. Equivalent to POSIX expression [^[:alnum:]_].	The expression **\w+\W\s\w+** matches the string to: bill but does not match to bill.
\s	Matches a whitespace character. Equivalent to POSIX expression [[:space:]].	The expression **\(\w\s\w\s\)** matches the string (a b) but does not match (ab) or (a,b.).
\S	Matches a nonwhitespace character. Equivalent to POSIX expression [^[:space:]].	The expression **\(\w\S\w\S\)** matches the strings (abde) and (a,b.) but does not match (a b d e).
\A	Matches the beginning of a string, in either single-line or multiline mode. Not equivalent to POSIX operator ^.	The expression **\AL** matches only the first L in the string Line1\nLine2\n (where \n is the newline character), in either single-line or multiline mode.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-5 (Cont.) PERL-Influenced Operators in Oracle SQL Regular Expressions

Operator Syntax	Description	Examples
\Z	Matches the end of a string, in either single-line or multiline mode. Not equivalent to POSIX operator $.	The expression **\s\Z** matches the last space in the string `L i n e \n` (where \n 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 $.	The expression **\s\z** matches the newline character (\n) in the string `L i n e \n`, in either single-line or multiline mode.
+?	Matches one or more occurrences of the preceding subexpression (nongreedy[1]).	The expression **\w+?x\w** matches **ab**xc in the string abxcxd (and the greedy expression **\w+x\w** matches **abxc**xd).
*?	Matches zero or more occurrences of the preceding subexpression (nongreedy[1]). Matches the empty string whenever possible.	The expression **\w*?x\w** matches xa in the string xaxbxc (and the greedy expression **\w*x\w** matches **xaxb**xc.
??	Matches zero or one occurrences of the preceding subexpression (nongreedy[1]). Matches the empty string whenever possible.	The expression **a??aa** matches aa in the string aaaa (and the greedy expression **a?aa** matches **a**aa).
{*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 aaaaa.
{*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 (?).

11.5 Using Regular Expressions in SQL Statements: Scenarios

Scenarios:

- Using a Constraint to Enforce a Phone Number Format
- Example: Enforcing a Phone Number Format with Regular Expressions
- Example: Inserting Phone Numbers in Correct and Incorrect Formats
- Using Back References to Reposition Characters

11.5.1 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.

Table 11-6 explains the elements of the regular expression in Example: Enforcing a Phone Number Format with Regular Expressions.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 11-6 Explanation of the Regular Expression Elements

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.
\)	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.

11.5.2 Example: Enforcing a Phone Number Format with Regular Expressions

When you create a table, you can enforce formats with regular expressions.

Example 11-1 creates a `contacts` table and adds a `CHECK` constraint to the `p_number` column to enforce this format model:

```
(XXX) XXX-XXXX
```

Example 11-1 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}$'))
);
```

11.5.3 Example: Inserting Phone Numbers in Correct and Incorrect Formats

The `INSERT INTO` SQL statement can be used to test how correct and incorrect formats work.

Example 11-2 shows some statements that correctly and incorrectly insert phone numbers into the `contacts` table.

Example 11-2 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');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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');
```

11.5.4 Using Back References to Reposition Characters

A back reference (described in Table 11-3) stores the referenced subexpression in a temporary buffer. Therefore, you can use back references to reposition characters, as in Example 11-3. For an explanation of the elements of the regular expression in Example 11-3, see Table 11-7.

Table 11-7 explains the elements of the regular expression in Example 11-3.

Table 11-7 Explanation of the Regular Expression Elements

Regular Expression Element	Description
^	Matches the beginning of the string.
$	Matches the end of the string.
(\S+)	Matches one or more nonspace characters. The parentheses are not escaped so they function as a grouping expression.
\s	Matches a whitespace character.
\1	Substitutes the first subexpression, that is, the first group of parentheses in the matching pattern.
\2	Substitutes the second subexpression, that is, the second group of parentheses in the matching pattern.
\3	Substitutes the third subexpression, that is, the third group of parentheses in the matching pattern.
,	Inserts a comma character.

Example 11-3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

12

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.

> **See Also**
>
> - *Oracle Database Concepts* for more information about indexes and index-organized tables
> - *Oracle Database Administrator's Guide* for more information about managing indexes
> - *Oracle Database SQL Tuning Guide* for more information about how indexes and clusters can enhance or degrade performance

Topics:

- Guidelines for Managing Indexes
- Managing Indexes
- When to Use Domain Indexes
- When to Use Function-Based Indexes

12.1 Guidelines for Managing Indexes

The summary of guidelines for managing the indexes are as follows:

- 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Consider cost before disabling or dropping constraints

> **See Also**
>
> *Oracle Database Administrator's Guide*

12.2 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**
>
> Creating Indexes for Use with Constraints

12.3 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).

> **See Also**
>
> - *Oracle Database Data Cartridge Developer's Guide* for conceptual background to help you decide when to build domain indexes
> - *Oracle Database Concepts* for information about domain indexes

12.4 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 computation is intensive). However:

- The database must also evaluate the index expression to process statements that do not use it.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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`.
> - Oracle Database treats descending indexes as if they were function-based indexes.

Topics:

- Advantages of Function-Based Indexes
- Disadvantages of Function-Based Indexes
- Example: Function-Based Index for Precomputing Arithmetic Expression
- Example: Function-Based Indexes on Object Column
- Example: Function-Based Index for Faster Case-Insensitive Searches
- Example: Function-Based Index for Language-Dependent Sorting

> **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
> - *Oracle Database SQL Language Reference* for information about `NVL`
> - *Oracle Database SQL Language Reference* for more information about creating index

12.4.1 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 instead of a full index scan.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 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 intensive computation the index expression gets, the index improves application performance.

- 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.

- 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 12-3) and the SQL function `NLSSORT` for linguistic-based sorts.

See Also

- *Oracle Database Concepts* for more information about index-range scan and index scan
- *Oracle Database SQL Language Reference* for more information about `ANALYZE` statement
- *Oracle Database Object-Relational Developer's Guide* for more information about function-based index
- Example: Function-Based Indexes on Object Column and *Oracle Database SQL Language Reference* for examples related to function-based indexes
- Example: Function-Based Index for Precomputing Arithmetic Expression
- Example: Function-Based Index for Language-Dependent Sorting for example related to `NLSSORT` SQL function

12.4.2 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` or `DBMS_STATS.GATHER_SCHEMA_STATS` .

- 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.
- A bitmapped function-based index cannot be a descending index.
- Index skip scanning is not supported in function-based indexes.
- 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_STATS`, `GATHER_TABLE_STATS`, `DBMS_STATS`, and `GATHER_SCHEMA_STATS`
- *Oracle Database SQL Language Reference* for more information about aggregate functions
- *Oracle Database SQL Language Reference* for more information about function-based index

12.4.3 Example: Function-Based Index for Precomputing Arithmetic Expression

You can create composite indexes to computer arithmetic expressions.

Example 12-1 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 12-1 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;
```

Note

This example uses composite indexes (indexes on multiple table columns).

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database Concepts* for information about fast full index scans
- *Oracle Database Concepts* for more information about composite indexes

12.4.4 Example: Function-Based Indexes on Object Column

In Example 12-2, 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 12-2 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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
```

12.4.5 Example: Function-Based Index for Faster Case-Insensitive Searches

Example 12-3 creates an index that allows faster case-insensitive searches in the `EMPLOYEES` table and then uses the index expression in a query.

Example 12-3 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.
```

12.4.6 Example: Function-Based Index for Language-Dependent Sorting

You can use the `NLSSORT` API for language-dependent sorting.

Example 12-4 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 12-4 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13

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.

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:

> **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.

> **See Also**
>
> - *Oracle Database Concepts* for information about data integrity and constraints
> - *Oracle Database Administrator's Guide* for more information about managing constraints
> - *Oracle Database SQL Language Reference* for the syntactic and semantic information about constraints
> - *Oracle Database SQL Language Reference* for more information about constraints on views
> - *Oracle Database Data Warehousing Guide* for information about using constraints in data warehouses
> - How the Correct Data Type Increases Data Integrity for more information about the role that data type plays in data integrity

Topics:

- Enforcing Business Rules with Constraints
- Enforcing Business Rules with Both Constraints and Application Code
- Creating Indexes for Use with Constraints

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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

13.1 Enforcing Business Rules with Constraints

Whenever possible, enforce business rules with constraints. 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 13-1 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 13-1 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

See Also

- *Oracle Database SQL Language Reference* for syntactic and semantic information about constraints
- *Oracle Database Concepts* for the complete list of advantages of integrity constraints
- *Oracle Database Concepts* for more information about using triggers to enforce business rules

13.2 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 13-2, assume that Example 13-1 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 13-2.

Example 13-2 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
END;
/

BEGIN
  add_employee ('Smith', 'H', 356, 20);
END;
/
```

Result:

```
Gender must be M or F.
```

13.3 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.

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.

See Also

- *Oracle Database Administrator's Guide* for more information about indexes associated with constraints
- *Oracle Database Administrator's Guide* for information about disabling and dropping constraints
- *Oracle Database Administrator's Guide* for information about creating indexes explicitly
- Using Indexes in Database Applications
- *Oracle Database SQL Language Reference* for information about creating indexes explicitly

13.4 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.

Use a `NOT NULL` constraint in both of these situations:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 13-3 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 13-3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

See Also

- *Oracle Database SQL Language Reference* for more information about NOT NULL constraint
- When to Use Default Column Values for more information about the usage of default column values
- UNIQUE and NOT NULL Constraints on the Foreign Key

13.5 When to Use Default Column Values

When an INSERT statement 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 while creating the table or when you alter the column using the ALTER TABLE statement.

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.

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 inserter (which need not be included in the definition

Copyright © 1999, 2025, Oracle and/or its affiliates.

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),
  inserter VARCHAR2(30) DEFAULT USER);
```

See Also

- *Oracle Database SQL Language Reference* for more information about the `INSERT` statement
- *Oracle Database SQL Language Reference* for more information about the `CREATE TABLE` statement
- *Oracle Database SQL Language Reference* for more information about the `ALTER TABLE` statement
- When to Use NOT NULL Constraints for information about the `NOT NULL` constraint

13.6 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.

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
- *Oracle Database SQL Language Reference* for information about sequences

Copyright © 1999, 2025, Oracle and/or its affiliates.

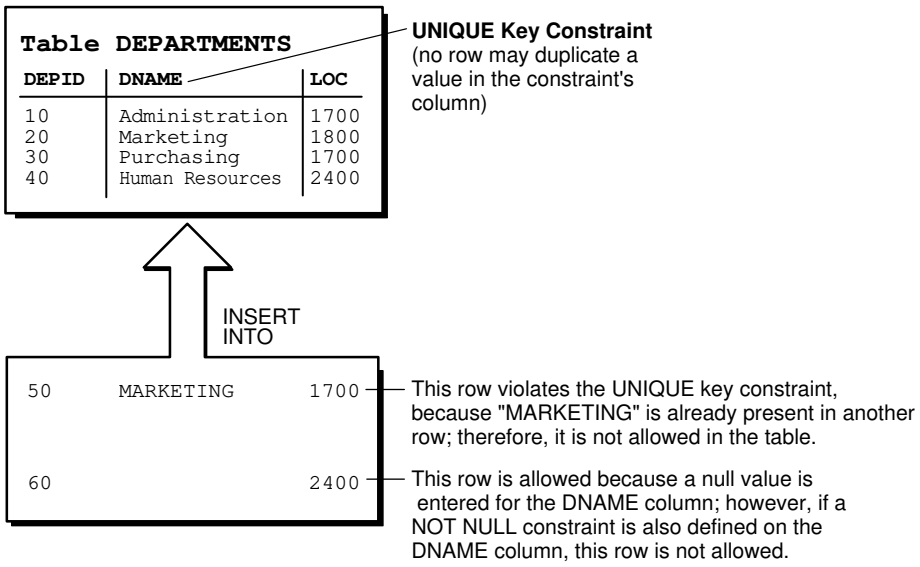
13.7 When to Use UNIQUE Constraints

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 13-1 shows a table with a `UNIQUE` constraint, a row that violates the constraint, and a row that satisfies it.

Figure 13-1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.8 Enforcing Referential Integrity with FOREIGN KEY Constraints

When two tables share one or more columns, you use 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`.

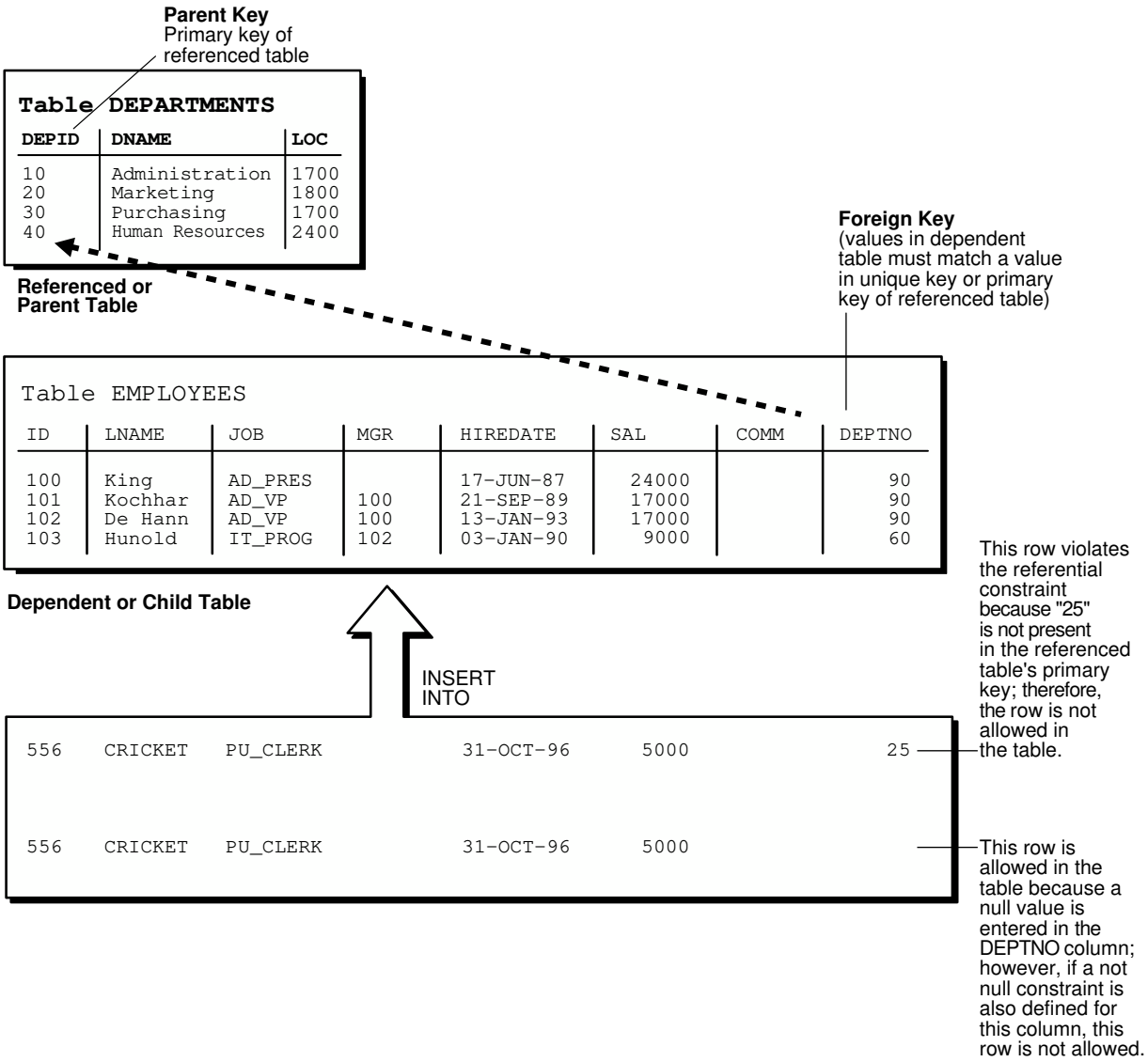
Designate one table as the **referenced or parent table** and the other as the **dependent or child table**. In the parent table, define either a `PRIMARY KEY` or `UNIQUE` constraint on the shared columns. In the child table, define a `FOREIGN KEY` constraint on the shared columns. The shared columns now comprise a **foreign key**. Defining additional constraints on the foreign key affects the parent-child relationship.

Figure 13-2 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 13-2 shows parent and child tables that share one column, a row that violates the `FOREIGN KEY` constraint, and a row that satisfies it.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 13-2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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
> - Defining Relationships Between Parent and Child Tables

13.8.1 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about triggers

13.8.2 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 13-2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

13.8.3 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.

13.8.4 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 13-4, the PRIMARY and FOREIGN keys of the table emp are created DEFERRABLE and then deferred.

Example 13-4 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

13.9 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **See Also**
>
> *Oracle Database SQL Language Reference* for more details and examples of integrity constraints

13.10 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

13.11 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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.

13.12 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**
>
> 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.

13.12.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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)

13.12.2 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*

13.12.3 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.

13.12.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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))
```

13.13 Using PRECHECK to Pre-validate a CHECK Constraint

A category of CHECK constraints has the property that can be pre-validated at the application client using JSON schema, before the data is sent to the database. Such constraints have the PRECHECK property, which make them precheckable. PRECHECK constraints enable cross-tier input data validation, both within the application client and the database using a single set of rules, which reside inside the database in the form of the CHECK constraints.

A constraint has the PRECHECK property when:

- The constraint has an equivalent JSON schema that preserves the semantics of the constraint.

  > **Note**
  >
  > Not all CHECK constraints can be expressed in JSON schema.

- The constraint was checked to ascertain if it has an equivalent JSON schema and this information was recorded inside the database.

  For newly defined tables, starting Oracle AI Database 26ai, all the corresponding CHECK constraints are checked to ascertain if they have the PRECHECK property when the constraints are defined, and the result is recorded. For existing tables with constraints, you can use the DDL statements provided in this section to check and set the constraint property to PRECHECK.

JSON schema is a well-established standard vocabulary for annotating and validating JSON data within applications. For applications that can understand and process JSON data, JSON schema is a viable option for client-side input validation.

Many user interfaces can easily convert data from input forms into various formats, including JSON. Data in JSON format from input forms can be validated against a JSON schema before sending it to the database. For an input table with constraints, the database can generate the corresponding JSON schema using the DBMS_JSON_SCHEMA.DESCRIBE() PL/SQL function. In the JSON schema that is generated by this function, CHECK constraints having the PRECHECK property are represented as sub-schemas. Hence, with JSON schema and PRECHECK constraints, you can successfully pre-validate data at the application client level.

Only a subset of SQL conditions that are used in CHECK constraints have an equivalent condition in the JSON schema vocabulary. Therefore, only a subset of CHECK constraints can have the PRECHECK property.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - Supported Conditions for JSON Schema Validation for a list of SQL conditions that have an equivalent in the JSON schema and are supported for PRECHECK JSON schema validation
> - JSON schema in *Oracle AI Database JSON Developer's Guide* for more information about JSON schema

13.13.1 PRECHECK Syntax and Definition

The PRECHECK keyword sets the PRECHECK property of the CHECK constraint. When the PRECHECK keyword is explicitly specified, the corresponding CHECK constraint is evaluated to ascertain whether an equivalent JSON schema exists. If an equivalent JSON schema exists, the PRECHECK property of the constraint is set and the DDL statement succeeds. Otherwise, the DDL statement fails with the error ORA-40544.

> **Note**
>
> You can use the PRECHECK constraint property only with the CHECK constraint.

Syntax for Defining CHECK Constraint with PRECHECK at Table Creation

```
CREATE TABLE <table_name> (
  <column_definition> CHECK (<condition>) [<constraint_state>] [PRECHECK | NOPRECHECK],
  CONSTRAINT <constraint_name> CHECK (<condition>)
    [<constraint_state>] [PRECHECK | NOPRECHECK]
)
```

Starting Oracle AI Database 26ai, for newly defined tables that do not explicitly specify the PRECHECK or NOPRECHECK keyword, all their corresponding CHECK constraints are evaluated to ascertain whether they have a JSON schema equivalent when the constraints are defined, and the result is recorded. When a constraint has a JSON Schema equivalent that preserves the semantics of the constraint, the PRECHECK property is set. Otherwise, the NOPRECHECK property is set. These constraint properties can be queried from the ALL|USER|DBA_CONSTRAINTS views. These views have the PRECHECK column with PRECHECK or NOPRECHECK value.

When the NOPRECHECK keyword is explicitly specified, the NOPRECHECK property is set for the constraint independently of whether a JSON schema equivalent exists; no evaluation takes place. For a constraint that has a JSON schema equivalent, you may want to mark it with NOPRECHECK when the constraint is not relevant to be validated on the client side. A NOPRECHECK constraint is not included in the JSON schema corresponding to the table.

As a restriction, other constraint states must precede the PRECHECK or NOPRECHECK property, whenever other constraint states are included.

Syntax for Defining CHECK Constraint with PRECHECK after Table Creation

```
ALTER TABLE <table_name>
  MODIFY CONSTRAINT <constraint_name> [<constraint_state>] [PRECHECK | NOPRECHECK]

ALTER TABLE <table_name>
  ADD CONSTRAINT <constraint_name> CHECK (<condition>) [<constraint_state>] [PRECHECK |
NOPRECHECK]
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Supported Conditions for JSON Schema Validation for a list of SQL conditions that have an equivalent in the JSON schema and are supported for `PRECHECK` JSON schema validation

13.13.1.1 Supported Conditions for JSON Schema Validation

The following SQL conditions are supported in the JSON schema vocabulary.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
`column >= <value>`	SQL condition translates into a column property with a `minimum`.	`Color NUMBER CHECK (Color >= 10)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "minimum" : 10 } ] }`
`column <= <value>`	SQL condition translates into a column property with a `maximum`.	`Color NUMBER CHECK (Color <= 20)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "maximum" : 20 } ] }`
`column > <value>`	SQL condition translates into a column property with an `exclusiveMinimum`.	`Color NUMBER CHECK (Color > 10)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "exclusiveMinimum" : 10 } ] }`

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
`column < <value>`	SQL condition translates into a column property with an `exclusiveMaximum`.	`Color NUMBER CHECK (Color < 20)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "exclusiveMaximum" : 20 } ] }`
`column = <val>`	SQL condition translates into a column property with a `const` value.	`Color NUMBER CHECK (Color = 10)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "const" : 10 } ] }`
`column <> <value>`	SQL condition translates into a column property with a negated (`not`) `const` value.	`Color NUMBER CHECK (Color <> 10)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "not" : { "const" : 10 } } ] }`

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
column BETWEEN val1 AND val2	SQL condition translates into a column property with a conjunction (allOf) of two conditions minimum and maximum.	Color NUMBER CHECK (Color between 10 and 20)	"COLOR" : { "extendedType" : "number", "allOf" : [{ "allOf" : [{ "minimum" : 10 }, { "maximum" : 20 }] }] }
column IN (<val1>, <val2>,... <valN>)	SQL condition translates into a column property with an enum.	Color NUMBER CHECK (Color IN (10, 15, 20))	"COLOR" : { "extendedType" : "number", "allOf" : [{ "enum" : [10, 15, 20] }] }
column = <val1> OR column=<val2> ... OR column=<valN> Equivalent with column IN (<val1>, <val2>,... <valN>)	SQL condition translates into a column property with an enum.	Color NUMBER CHECK (Color =10 OR Color=15 OR Color =20))	"COLOR" : { "extendedType" : "number", "allOf" : [{ "enum" : [10, 15, 20] }] }

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
`MOD(col, value) = 0`	SQL condition translates into a column property with a `multipleOf`.	`Color NUMBER CHECK (MOD(Color, 2) = 0)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "multipleOf" : 2 } ] }`
`REGEXP_LIKE(col, <regex_pattern>)`	SQL condition translates into a column property with a `pattern`.	`Name VARCHAR2(50) CHECK (REGEXP_LIKE(Name, '^Product'))`	`"NAME" : { "extendedType" : "string", "maxLength" : 50, "allOf" : [ { "pattern" : "^Product" } ] }`
`NOT REGEXP_LIKE(col, <regex_pattern>)`	SQL condition translates into a column property with a negated (`not`) `pattern`.	`Name VARCHAR2(50) CHECK (NOT REGEXP_LIKE(Name, '^Product'))`	`"NAME" : { "extendedType" : "string", "maxLength" : 50, "allOf" : [ { "not" : { "pattern" : "^Product" } } ] }`

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
`LENGTH(column) <= <length>`	SQL condition translates into a column property with a `maxLength`.	`Name VARCHAR2(50) CHECK (LENGTH(Name) <= 40)`	`"NAME" : { "extendedType" : "string", "maxLength" : 50, "allOf" : [ { "maxLength" : 40 } ] }`
`LENGTH(column) >= <length>`	SQL condition translates into a column property with a `minLength`.	`Name VARCHAR2(50) CHECK (LENGTH(Name) >= 10)`	`"NAME" : { "extendedType" : "string", "maxLength" : 50, "allOf" : [ { "minLength" : 10 } ] }`
`column IS NOT NULL`	SQL condition translates into adding the column name to the `required` array of the JSON Schema (meaning that the corresponding column property is required).	`Name VARCHAR2(50) NOT NULL`	`"required" : [ "NAME" ]`

`column IS JSON`	SQL condition translates into a column property corresponding to any JSON record.	Example 1: `jcol JSON CHECK (jcol IS JSON)`
		Example 2: `jcol BLOB CHECK (jcol IS JSON format oson (size limit 32m))`
			Example 1:
			`"JCOL" : { }`
			Example 2:
			`"JCOL" : { "extendedType" : [ "null", "binary" ] }`

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
`column IS JSON VALIDATE USING`	SQL condition will use the provided schema for the column validation. It translates to a column property with the provided schema.	`jcol JSON CHECK (jcol IS JSON VALIDATE USING '{ "type": ["array", "object"] }')`	`"JCOL" : { "allOf" : [ { "type" : [ "array", "object" ] } ] }`
`AND`	SQL condition translates into a column property with a conjunction (`allOf`) of multiple conditions.	`Color NUMBER CHECK (Color >= 10 AND Color <=20)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "allOf" : [ { "minimum" : 10 }, { "maximum" : 20 } ] } ] }`

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL Condition	Equivalent in JSON Schema	SQL Example	JSON Schema Example
OR	SQL condition translates into a column property with a disjunction (anyOf) of multiple conditions.	`Color NUMBER CHECK (Color >= 10 OR Color <=20)`	`"COLOR" : { "extendedType" : "number", "allOf" : [ { "anyOf" : [ { "minimum" : 10 }, { "maximum" : 20 } ] } ] }`
NOT	SQL condition translates into a column property with a negated condition (not).	Supported in conjunction with a given operator. For example, `NOT REGEXP_LIKE(col, <regex_pattern>)` or `<>`.	`"NAME" : { "extendedType" : "string", "maxLength" : 50, "allOf" : [ { "not" : { "pattern" : "^Product" } } ] }`

13.13.2 Enabling PRECHECK for a New Relational Table

For newly defined tables, starting Oracle AI Database 26ai, all their corresponding CHECK constraints are evaluated to ascertain whether they have a JSON schema equivalent when the constraints are defined, and the result is recorded.

In the following example, a Product table is created. The queried constraint information that follows shows that the SYS_C008515, SYS_C008516, SYS_C008517, and SYS_C008518 constraints have the PRECHECK property, whereas the MIXEDCOL constraint has the NOPRECHECK property. This is because the corresponding CHECK condition for the MIXEDCOL constraint is an inequality among the two columns that do not have a JSON schema equivalent.

```
CREATE TABLE Product(
      Id NUMBER NOT NULL PRIMARY KEY,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      Name VARCHAR2(50) CHECK (regexp_like(Name, '^Product')),
      Category VARCHAR2(10) NOT NULL CHECK (CATEGORY IN ('Home', 'Apparel')),
      Price NUMBER CHECK (mod(price,4) = 0 and 10 < price),
      Description VARCHAR2(50) CHECK (Length(Description) <= 40),
      Created_At DATE,
      Updated_At DATE,
      CONSTRAINT MIXEDCOL CHECK (Created_At > Updated_At)
    );
```

You can query the constraint information, as follows:

```
SELECT constraint_name, search_condition_vc, precheck
 FROM all_constraints
 WHERE table_name='PRODUCT';
```

The output is:

```
CONSTRAINT_NAME      SEARCH_CONDITION_VC                      PRECHECK
-------------------- ---------------------------------------- ----------
SYS_C008513          "ID" IS NOT NULL
SYS_C008514          "CATEGORY" IS NOT NULL
SYS_C008515          regexp_like(Name, '^Product')            PRECHECK
SYS_C008516          CATEGORY IN ('Home', 'Apparel')          PRECHECK
SYS_C008517          mod(price,4) = 0 and 10 < price          PRECHECK
SYS_C008518          Length(Description) <= 40                PRECHECK
MIXEDCOL             Created_At > Updated_At                  NOPRECHECK
SYS_C008520
```

If you would like to be informed in advance when a particular constraint cannot be set to `PRECHECK`, you can explicitly use the `PRECHECK` keyword against the constraint. For instance, you can specify `PRECHECK` for the `MIXEDCOL` constraint, as follows:

```
CREATE TABLE Product(
      Id NUMBER NOT NULL PRIMARY KEY,
      Name VARCHAR2(50) CHECK (regexp_like(Name, '^Product')),
      Category VARCHAR2(10) NOT NULL CHECK (CATEGORY IN ('Home', 'Apparel')),
      Price NUMBER CHECK (mod(price,4) = 0 and 10 < price),
      Description VARCHAR2(50) CHECK (Length(Description) <= 40),
      Created_At DATE,
      Updated_At DATE,
      CONSTRAINT MIXEDCOL CHECK (Created_At > Updated_At) PRECHECK
    );
```

The DDL statement fails, returning an `ORA-40544` error:

```
ERROR at line 9:
ORA-40544: CHECK expression of 'MIXEDCOL' constraint not possible to use as PRECHECK
condition
```

The JSON schema corresponding to the `PRODUCT` table that is created using the first `CREATE TABLE` statement is listed in the following example. The constraints with the `PRECHECK` property have sub-schemas corresponding to the `CHECK` constraint conditions within the corresponding columns (see the "allOf" entries), whereas the `MIXEDCOL` constraint with no equivalent JSON schema is listed in the "dbNoPrecheck" array.

```
SELECT dbms_json_schema.DESCRIBE('PRODUCT');

DBMS_JSON_SCHEMA.DESCRIBE('PRODUCT')
--------------------------------------------------------------------------------
{
  "title" : "PRODUCT",
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
"dbObject" : "SYS.PRODUCT",
"type" : "object",
"dbObjectType" : "table",
"properties" :
{
  "ID" :
  {
    "extendedType" : "number"
  },
  "NAME" :
  {
    "extendedType" :
    [
    "null",
    "string"
    ],
    "maxLength" : 50,
    "allOf" :
    [
    {
      "pattern" : "^Product"
    }
    ]
  },
  "CATEGORY" :
  {
    "extendedType" : "string",
    "maxLength" : 10,
    "allOf" :
    [
    {
      "enum" :
      [
        "Home",
        "Apparel"
      ]
    }
    ]
  },
  "PRICE" :
  {
    "extendedType" :
    [
    "null",
    "number"
    ],
    "allOf" :
    [
    {
      "allOf" :
      [
        {
          "multipleOf" : 4
        },
        {
          "exclusiveMinimum" : 10
        }
      ]
    }
    ]
  },
  "DESCRIPTION" :
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    {
      "extendedType" :
      [
      "null",
      "string"
      ],
      "maxLength" : 50,
      "allOf" :
      [
      {
        "maxLength" : 40
      }
      ]
    },
    "CREATED_AT" :
    {
      "extendedType" :
      [
      "null",
      "date"
      ]
    },
    "UPDATED_AT" :
    {
      "extendedType" :
      [
      "null",
      "date"
      ]
    }
  },
  "required" :
  [
    "ID",
    "CATEGORY"
  ],
  "dbNoPrecheck" :
  [
    {
      "dbConstraintName" : "MIXEDCOL",
      "dbConstraintExpression" : "Created_At > Updated_At"
    }
  ],
  "dbPrimaryKey" :
  [
    "ID"
  ]
}
```

13.13.3 Enabling PRECHECK for an Existing Table

You can enable the `PRECHECK` property for an existing table. The following example uses an existing table called `HR.EMPLOYEES`.

```
SQL> DESC HR.EMPLOYEES;

 Name                      Null?      Type
 ------------------------- ---------- -----------------------------------
 EMPLOYEE_ID               NOT NULL   NUMBER(6)
 FIRST_NAME                           VARCHAR2(20)
 LAST_NAME                 NOT NULL   VARCHAR2(25)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
EMAIL                     NOT NULL   VARCHAR2(25)
PHONE_NUMBER                         VARCHAR2(20)
HIRE_DATE                 NOT NULL   DATE
JOB_ID                    NOT NULL   VARCHAR2(10)
SALARY                               NUMBER(8,2)
COMMISSION_PCT                       NUMBER(2,2)
MANAGER_ID                           NUMBER(6)
DEPARTMENT_ID                        NUMBER(4)
```

This table already has a `CHECK` constraint `EMP_SALARY_MIN`. The `PRECHECK` column in `ALL_CONSTRAINTS` is `NULL` for this constraint. A `NULL` value means that the `PRECHECK` property was not yet initialized. You can set the `PRECHECK` property with the following DDL statement.

```
ALTER TABLE HR.EMPLOYEES
  MODIFY CONSTRAINT EMP_SALARY_MIN PRECHECK;
```

You can query the constraint information, as follows:

```
SELECT CONSTRAINT_NAME, SEARCH_CONDITION_VC, PRECHECK
  FROM ALL_CONSTRAINTS
  WHERE TABLE_NAME='EMPLOYEES' AND CONSTRAINT_NAME='EMP_SALARY_MIN';
```

The output is:

```
CONSTRAINT_NAME      SEARCH_CONDITION_VC                     PRECHECK
-------------------- --------------------------------------- ----------
EMP_SALARY_MIN       salary > 0                              PRECHECK
```

If this constraint is not relevant for the client-side validation, and you do not want it to be included in the corresponding JSON schema, you can set the `NOPRECHECK` property instead of the `PRECHECK` property, as follows:

```
ALTER TABLE HR.EMPLOYEES
  MODIFY CONSTRAINT EMP_SALARY_MIN NOPRECHECK;
```

You can also add new constraints, as in the following example. In the example, a new constraint is added without specifying the `PRECHECK` keyword. The constraint is implicitly set to `PRECHECK` because it has a JSON schema equivalent.

```
ALTER TABLE HR.EMPLOYEES
  ADD CONSTRAINT EMP_COMMISSION_PCT_MIN CHECK (COMMISSION_PCT >= 0.1);
```

The output is:

```
Table altered.
```

You can query the newly added constraint information, as follows:

```
SELECT CONSTRAINT_NAME, SEARCH_CONDITION_VC, PRECHECK
  FROM ALL_CONSTRAINTS
  WHERE TABLE_NAME='EMPLOYEES' AND CONSTRAINT_NAME='EMP_COMMISSION_PCT_MIN';
```

The output is:

```
CONSTRAINT_NAME           SEARCH_CONDITION_VC                     PRECHECK
------------------------- --------------------------------------- ----------
EMP_COMMISSION_PCT_MIN    COMMISSION_PCT >= 0.1                   PRECHECK
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

In the following example, the constraint is implicitly set to NOPRECHECK since there is no JSON schema equivalent for the constraint that has two columns in the check condition.

```
ALTER TABLE HR.EMPLOYEES
  ADD CONSTRAINT EMP_MAX_BONUS CHECK ((SALARY * COMMISSION_PCT) < 6000);
```

The output is:

```
Table altered.
```

You can query the added constraint information, as follows:

```
SELECT CONSTRAINT_NAME, SEARCH_CONDITION_VC, PRECHECK
  FROM ALL_CONSTRAINTS
  WHERE TABLE_NAME='EMPLOYEES' AND CONSTRAINT_NAME='EMP_MAX_BONUS';
```

The output is:

```
CONSTRAINT_NAME           SEARCH_CONDITION_VC                     PRECHECK
------------------------- --------------------------------------- ----------
EMP_MAX_BONUS             (SALARY * COMMISSION_PCT) < 6000        NOPRECHECK
```

Using the PRECHECK keyword in this example raises an error, and the DDL statement fails, as follows:

```
ALTER TABLE HR.EMPLOYEES
  ADD CONSTRAINT EMP_MAX_BONUS CHECK ((SALARY * COMMISSION_PCT) < 6000) PRECHECK;
```

The output is:

```
ERROR at line 1:
ORA-40544: CHECK expression of 'EMP_MAX_BONUS' constraint not possible to use as
PRECHECK condition
```

13.13.4 Guidelines for Using PRECHECK

The PRECHECK functionality provides you the option to validate the JSON data in advance on the client side before the data reaches the database. Using the PRECHECK functionality, you can restrict invalid data from being sent to the database.

Use PRECHECK with CHECK constraints in the following modes when altering tables or creating new tables.

PRECHECK (PRECHECK + ENABLE)

Using the PRECHECK and the ENABLE states together, you can have the constraints prechecked for an existing table or a new table, whereby the constraints are prechecked before the JSON data is sent to the database on the client side. Subsequently, the constraints are checked within the database as well.

PRECHECK + DISABLE for New Constraints

You may want to define new constraints for application-level checks that are currently not defined as constraints inside the database. You can use application-level logic to check the input data, which is totally independent of database constraints. But, using the PRECHECK feature with the DISABLE state, you can declare these checks as disabled constraints and use the corresponding JSON Schema to continue the validation within the application client. Using the PRECHECK feature enables you to define these new constraints inside the database and

Copyright © 1999, 2025, Oracle and/or its affiliates.

share them with other developers. Examples for such checks include checking validity of email addresses, phone numbers, state within an address, and dates.

PRECHECK + DISABLE for Existing Constraints that were Earlier in the ENABLE State

Using the `PRECHECK` and `DISABLE` together, you can validate data within the application layer and ensure consistency of its schema with the table definition inside the database. The corresponding constraints are no longer validated within the database. When using this mode, the schema changes by other developers may affect the consistency of your own local schema. You must ensure that you have a mechanism to maintain a consistent schema before using it for prechecking data. Since the data cannot be checked inside the database, this mode is not recommended when there is a loose coupling between the developer and the database (the database is not managed by the developer and they are not notified about the schema changes to the database).

13.14 Examples of Defining Constraints

Example 13-5 and Example 13-6 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 13-5 creates tables and their constraints at the same time, using the `CREATE TABLE` statement.

Example 13-5 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 13-6 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 13-6 Defining Constraints with the ALTER TABLE Statement

```
-- Create tables without constraints:

DROP TABLE DeptTab;
CREATE TABLE DeptTab (
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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

13.14.1 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.

> **See Also**
>
> Privileges Required to Create FOREIGN KEY Constraints

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.14.2 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" for examples of static data dictionary views

13.15 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

13.15.1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

13.15.2 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 13-7.

Example 13-7 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**
>
> Fixing Constraint Exceptions, for more information about rows that violate constraints

13.15.3 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 13-8.

Example 13-8 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 */
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

13.15.4 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 13-9.

Example 13-9 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**
>
> Fixing Constraint Exceptions, for more information about rows that violate constraints

13.15.5 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 13-10.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 13-10 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;
```

13.15.6 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 Managing FOREIGN KEY Constraints

13.15.7 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**
>
> 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database Administrator's Guide* for more information about responding to constraint exceptions

13.16 Modifying Constraints

Starting with Oracle8*i*, you can modify an existing constraint with the `MODIFY CONSTRAINT` clause, as in Example 13-11.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about the parameters you can modify

Example 13-11 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.17 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 13-12 shows how to find the system-generated name of a constraint and change it.

Example 13-12 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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.18 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 13-13 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

13.19 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.19.1 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.

13.19.2 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.

13.19.3 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.

13.19.4 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 their tables.

13.19.5 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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);
  ```

13.20 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 13-14.

> **See Also**
>
> *Oracle Database Reference* for information about `*_CONSTRAINTS` and `*_CONS_COLUMNS`

Example 13-14 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

- Coding PL/SQL Subprograms and Packages
- Using PL/Scope
- Using the PL/SQL Hierarchical Profiler
- Using PL/SQL Basic Block Coverage to Maintain Quality
- Developing PL/SQL Web Applications
- Using Continuous Query Notification (CQN)

> **See Also**
>
> *Oracle AI Database PL/SQL Language Reference* for a complete description of PL/SQL

Copyright © 1999, 2025, Oracle and/or its affiliates.

14

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
- Deprecating Packages, Subprograms, and Types
- 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

14.1 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.

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.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

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.

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.

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.

A `BEFORE UPDATE` trigger can fire multiple times due to internal retries. Consider this while designing your applications. If you use a normal trigger, any work done by the trigger is rolled back when there is a retry. However if you define a trigger using autonomous transactions, then any work done by the trigger is not rolled back.

See Also

- *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL subprograms
- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL blocks
- *Oracle Database PL/SQL Language Reference* for more information about subprograms
- Overview of PL/SQL Units for information about PL/SQL units
- Developing Applications with Multiple Programming Languages for information about external subprograms
- *Oracle Database PL/SQL Language Reference* for more information about overloaded subprograms
- *Oracle Database PL/SQL Language Reference* for more information about triggers

14.2 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..

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 the accessor 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.

> **Note**
>
> Before you create your own package, check *Oracle AI Database PL/SQL Packages and Types Reference* to see if Oracle supplies a package with the functionality that you need.

> **See Also**
>
> - Overview of PL/SQL Units for information about PL/SQL units
> - *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL packages
> - *Oracle Database PL/SQL Language Reference* for the reasons to use packages

14.3 Overview of PL/SQL Units

A PL/SQL unit is one of these:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

The `AUTHID` property of a PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at runtime.

See Also

- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL units and compilation parameters
- *Oracle Database PL/SQL Language Reference* for more information about `AUTHID` property

14.3.1 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 14-1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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`

Example 14-1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.
```

14.4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.4.1 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
- Invoking Stored PL/SQL Subprograms

14.4.2 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`
- `CREATE PROCEDURE`
- `CREATE PACKAGE`
- `CREATE PACKAGE BODY`

  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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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
```

Alternatively, you can create and run the DDL statements using SQL Developer.

> **See Also**
>
> - *SQL*Plus User's Guide and Reference* for information about runnings scripts in SQL*Plus
> - *Oracle SQL Developer User's Guide* for information about SQL Developer
> - *Oracle Database PL/SQL Language Reference* for more information about the following functions
>   - `CREATE FUNCTION`
>   - `CREATE PROCEDURE`
>   - `CREATE PACKAGE`
>   - `CREATE PACKAGE BODY`

14.4.3 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.4.4 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.

> **See Also**
>
> Using the Correct and Most Specific Data Type

Topics:

- PL/SQL Scalar Data Types
- PL/SQL Composite Data Types
- Abstract Data Types

14.4.4.1 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

14.4.4.1.1 SQL Data Types

The PL/SQL data types include the SQL data types.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - Using SQL Data Types in Database Applications for information about how to use the SQL data types in database applications
> - *Oracle Database PL/SQL Language Reference* for more information about SQL data types

14.4.4.1.2 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `BOOLEAN` data type

14.4.4.1.3 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `PLS_INTEGER` data type

14.4.4.1.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `REF CURSOR` data type and cursor variables

14.4.4.1.5 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

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about user-defined PL/SQL subtypes

14.4.4.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about PL/SQL composite data types

14.4.4.3 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`.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about ADTs

14.4.5 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).

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.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database PL/SQL Language Reference* for general information about cursor variables, and query set result processing with implicit and explicit cursors
- *Oracle Database PL/SQL Language Reference* for more information about how to limit the number of rows and the collection size in bulk collect statements
- *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
- Returning Large Amounts of Data from a Function
- *Oracle Database Concepts* for more information about PGA

14.4.5.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

14.4.5.2 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.

> **See Also**
>
> *Oracle Database SQL Tuning Guide* for instructions for producing `TKPROF` reports

14.4.5.2.1 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_CURSORS`, 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 14-2, 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 14-2 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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
```

14.4.5.2.2 Multiple-Row-Fetching Penalty for Cursor Variable

Example 14-3 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 14-3 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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
```

14.4.5.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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.

> **See Also**
>
> - *Oracle Database PL/SQL Language Reference* for more information about `DBMS_SQL.RETURN_RESULT` procedure
> - *Oracle Database PL/SQL Language Reference* for information about using `DBMS_SQL` procedures for dynamic SQL

14.4.6 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

14.4.7 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about the PL/SQL function result cache

14.4.8 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.

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
> - *Oracle Database PL/SQL Language Reference* for more information about parallel DML statements

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.4.8.1 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 14-4 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `FORALL` statement

Example 14-4 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;
/
```

14.4.8.2 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `BULK COLLECT` clause

The PL/SQL block in Example 14-5 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 14-5 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;
/
```

14.4.8.3 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about use the `BULK COLLECT` clause with the `RETURNING INTO` clause

The PL/SQL block in Example 14-6 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 14-6 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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;
/
```

14.4.9 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.

> **See Also**
>
> - Understanding Schema Object Dependency for information about schema object dependency
> - *Oracle Database PL/SQL Language Reference* for more information about dynamic SQL
> - *Oracle Database PL/SQL Language Reference* for more information about static SQL

14.5 Altering PL/SQL Subprograms and Packages

To alter the name of a stored standalone subprogram or package, you must drop it 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **See Also**
>
> - Dropping PL/SQL Subprograms and Packages
> - *Oracle Database PL/SQL Language Reference* for information about `ALTER` statements

14.6 Deprecating Packages, Subprograms, and Types

As of Oracle Database 12*c* Release 2 (12.2), you can use the `DEPRECATE` pragma to communicate that a package, subprogram, or type has been deprecated or superseded by a new interface. When a unit is compiled that makes a reference to a deprecated element, a compilation warning is issued. To mark a PL/SQL unit as deprecated, use the `DEPRECATE` pragma.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about `DEPRECATE` pragma

14.7 Dropping PL/SQL Subprograms and Packages

To drop stored standalone subprograms, use these statements:

- `DROP FUNCTION`
- `DROP PROCEDURE`

To drop a package (specification and body) or only its body, use the `DROP PACKAGE` statement.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database PL/SQL Language Reference* for more information about `DROP FUNCTION`
- *Oracle Database PL/SQL Language Reference* for more information about `DROP PROCEDURE`
- *Oracle Database PL/SQL Language Reference* `DROP PACKAGE`

14.8 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.

Note

To compile Java packages and classes for native execution, use the `ncomp` tool.

See Also

- *Oracle Database PL/SQL Language Reference* for more information about compiling PL/SQL units for native execution
- *Oracle Database Java Developer's Guide*

14.9 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.

When you invoke a subprogram owned by another user:

- You must include the name of the owner in the invocation. For example:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

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 AUTHID property, 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
- 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

14.9.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about system and object privileges

14.9.2 Invoking a Subprogram Interactively from Oracle Tools

You can invoke a subprogram interactively from an Oracle Database tool, such as SQL*Plus.

> **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

Example 14-7 uses SQL*Plus to create a procedure and then invokes it in two different ways.

Some interactive tools allow you to create session variables, which you can use for the duration of the session. Using SQL*Plus, Example 14-8 creates, uses, and prints a session variable.

Example 14-7 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Result:

```
PL/SQL procedure successfully completed.
```

Example 14-8 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
```

14.9.3 Invoking a Subprogram from Another Subprogram

A subprogram or a trigger can invoke another stored subprogram. In Example 14-9, the procedure `print_mgr_name` invokes the procedure `print_emp_name`.

Recursive subprogram invocations are allowed (that is, a subprogram can invoke itself).

Example 14-9 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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
```

14.9.4 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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 Dependencies Among Local and Remote Database Procedures.

Topics:

- Synonyms for Remote Subprograms
- Transactions That Invoke Remote Subprograms

> **See Also**
>
> - Dependencies Among Local and Remote Database Procedures
> - *Oracle Database PL/SQL Language Reference* for information about handling errors in subprograms

14.9.4.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

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 Concepts* for examples of public synonym
- *Oracle Database SQL Language Reference* for information about the `CREATE SYNONYM` statement
- *Oracle Database SQL Language Reference* for information about the `GRANT` statement

14.9.4.2 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 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

14.10 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.
- `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*.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

See Also

- *Oracle Database SQL Language Reference* for more information about `SELECT` statement
- *Oracle Database PL/SQL Language Reference* for general information about invoking subprograms, including passing parameters

14.10.1 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).

14.10.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

14.10.3 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 14-10 satisfies the preceding requirements.

Example 14-10 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;
/
```

14.10.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Some side effects are not allowed in a function invoked from a SQL query or DML statement.

Before Oracle Database 8*g* 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.

Topics:

- Restrictions on Functions Invoked from SQL Statements
- PL/SQL Functions Invoked from Parallelized SQL Statements
- PRAGMA RESTRICT_REFERENCES (deprecated)

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for information about `DETERMINISTIC` and `PARALLEL_ENABLE` optimizer hints

14.10.4.1 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
- 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **See Also**
>
> Autonomous Transactions

14.10.4.2 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 8*g* 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 8*g* 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 8*g* Release 1 (8.1), the database recognized the function as parallelizable.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.10.4.3 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 Specifying the Assertion TRUST.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about `PRAGMA RESTRICT_REFERENCES`

Topics:

- Specifying the Assertion TRUST
- Differences between Static and Dynamic SQL Statements

14.10.4.3.1 Example: PRAGMA RESTRICT_REFERENCES

You can use the `PRAGMA RESTRICT_REFERENCES` clause when you create a PL/SQL package.

Example 14-11 creates a function that neither reads nor writes database or package state, and asserts that is has the maximum purity level.

Example 14-11 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

14.10.4.3.2 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 14-12) or the C or Java subprogram (as in Example 14-13), because the PL/SQL compiler cannot check a C or Java subprogram for violations at runtime.

Example 14-12 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 14-13 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

14.10.4.3.3 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';
```

14.11 Analyzing and 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`.
- Analyzing the program and its execution in greater detail by running PL/Scope, the PL/SQL hierarchical profiler, or a debugger

Topics:

- PL/Scope
- PL/SQL Hierarchical Profiler
- Debugging PL/SQL and Java

> **See Also**
>
> - *Oracle Database PL/SQL Language Reference* for information about handling errors in PL/SQL subprograms and packages
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_OUTPUT` package

14.11.1 PL/Scope

PL/Scope lets you develop powerful and effective PL/Scope source code tools that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

For more information about PL/Scope, see Using PL/Scope.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.11.2 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.

See Also

Using the PL/SQL Hierarchical Profiler for more information about PL/SQL hierarchical profiler

14.11.3 Debugging PL/SQL and Java

PL/SQL and Java code in the database can be debugged using Oracle SQL Developer, Oracle JDeveloper, and various third-party tools. The DBMS_DEBUG_JDWP package is used to establish connections between a database session and these debugger programs.

It is possible to investigate a problem occurring in a long running test, or in a production environment by connecting to the session to debug from another session. While debugging a session, it is possible to inspect the state of in-scope variables and to examine the database state as the session being debugged sees it during an uncommitted transaction. When stopped at a breakpoint, it is possible for the debugging user to issue SQL commands, and run PL/SQL code invoking stored PL/SQL subprograms in an anonymous block if necessary.

See Also

- *Oracle SQL Developer User's Guide* for more information about running and debugging functions and procedures
- *Oracle Database Java Developer's Guide* for information about using the Java Debug Wire Protocol (JDWP) PL/SQL Debugger
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_DEBUG_JDWP` package
- *Oracle Database Reference* for information about the `V$PLSQL_DEBUGGABLE_SESSIONS` view

14.11.3.1 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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> The PL/SQL compiler never generates debug information for code hidden with the PL/SQL `wrap` utility.

> **See Also**
>
> - *Oracle Database PL/SQL Language Reference* for information about the `wrap` utility
> - Overview of PL/SQL Units for information about PL/SQL units
> - *Oracle Database Reference* for more information about `PLSQL_OPTIMIZE_LEVEL`

14.11.3.2 Privileges for Debugging PL/SQL and Java Stored Subprograms

For a session to connect to a debugger, the effective user at the time of the connect operation must have the `DEBUG CONNECT SESSION` , `DEBUG CONNECT ANY`, or appropriate `DEBUG CONNECT ON USER` 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 not included.

> **Caution**
>
> Granting the `DEBUG ANY PROCEDURE` privilege, or granting the `DEBUG` privilege on any object owned by `SYS`, grants complete rights to the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
- *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_DEBUG_JDWP package security model

14.12 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 reinstantiated 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 reinstantiated. To reinstantiate the package, the client session must either reconnect to the database or recompile the package.

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.

14.13 Example: Raising an ORA-04068 Error

You can use the `RAISE` clause to raise exceptions.

In Example 14-14, 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 14-14 Raising ORA-04068

```
PROCEDURE p IS
  package_exception EXCEPTION;
  PRAGMA EXCEPTION_INIT (package_exception, -6508);
BEGIN
 ...
EXCEPTION
  WHEN package_exception THEN
    RAISE;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
END;
/
```

14.14 Example: Trapping ORA-04068

You can use the `RAISE` statement in a package definition to trap errors.

In Example 14-15, 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 reinstantiated.

Example 14-15 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

15

Using PL/Scope

PL/Scope lets you develop powerful and effective PL/Scope source code tools that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

PL/Scope is intended for application developers, and is typically used in a development database environment.

> **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:

- Overview of PL/Scope
- Privileges Required for Using PL/Scope
- Specifying Identifier and Statement Collection
- How Much Space is PL/Scope Data Using?
- Viewing PL/Scope Data
- Overview of Data Dictionary Views Useful to Manage PL/SQL Code
- Sample PL/Scope Session

15.1 Overview of PL/Scope

PL/Scope is a compiler-driven tool that collects PL/SQL and SQL identifiers as well as SQL statements usage in PL/SQL source code.

PL/Scope collects PL/SQL identifiers, SQL identifiers, and SQL statements metadata 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.

Starting with Oracle Database 12*c* Release 2 (12.2), PL/Scope has been enhanced to report on the occurrences of static SQL, and dynamic SQL call sites in PL/SQL units. The call site of the native dynamic SQL (EXECUTE IMMEDIATE, OPEN CURSOR FOR) and DBMS_SQL calls are collected. Dynamic SQL statements are generated at execution time, so only the call sites can be collected at compilation time. The collected data in the new DBA_STATEMENTS view can be queried along with the other data dictionary views to help answer questions about the scope of changes required for programming projects, and performing code analysis. It is also useful to identify the source of SQL statement not performing well. PL/Scope provides insight into dependencies between tables, views and the PL/SQL units. This level of details can be used as a migration assessment tool to determine the extent of changes required.

Copyright © 1999, 2025, Oracle and/or its affiliates.

PL/Scope can help you answer questions such as :

- Where and how a column x in table y is used in the PL/SQL code?
- Is the SQL in my application PL/SQL code compatible with TimesTen?
- What are the constants, variables and exceptions in my application that are declared but never used?
- Is my code at risk for SQL injection?
- What are the SQL statements with an optimizer hint coded in the application?
- Which SQL has a BULK COLLECT clause ? Where is the SQL called from ?

15.2 Privileges Required for Using PL/Scope

By default, PUBLIC has SELECT privileges on various system tables and views, and EXECUTE privileges on various PL/SQL objects.

The PL/Scope data is available in the DBA_IDENTIFIERS and DBA_STATEMENTS data dictionary views. The user must have the privileges to query data in these views.

The following privileges have been granted on these relevant views.

View Name	Privilege Granted to Role
USER_IDENTIFIERS	READ to PUBLIC
ALL_IDENTIFIERS	READ to PUBLIC
DBA_IDENTIFIERS	SELECT to SELECT_CATALOG_ROLE
USER_STATEMENTS	READ to PUBLIC
ALL_STATEMENTS	READ to PUBLIC
DBA_STATEMENTS	SELECT to SELECT_CATALOG_ROLE

A database administrator can verify the list of privileges on these views by using a query similar to the following:

```
SELECT *
FROM   SYS.DBA_TAB_PRIVS
WHERE  GRANTEE = 'PUBLIC'
AND TABLE_NAME IN
('ALL_IDENTIFIERS','USER_IDENTIFIERS','ALL_STATEMENTS','USER_STATEMENTS');
```

15.3 Specifying Identifier and Statement Collection

By default, PL/Scope does not collect data for identifiers and statements in the PL/SQL source program. To enable and control what is collected, set the PL/SQL compilation parameter PLSCOPE_SETTINGS.

Starting with Oracle Database 12*c* Release 2 (12.2), the `PLSCOPE_SETTINGS` has a new syntax that offers more controls and options to collect identifiers and SQL statements metadata. The metadata is collected in the static data dictionary views `DBA_IDENTIFIERS` and `DBA_STATEMENTS`.

To collect PL/Scope 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'`. The possible values for the `IDENTIFIERS` clause are : `ALL`, `NONE` (default), `PUBLIC`, `SQL`, and `PLSQL`. New SQL identifiers are introduced for : `ALIAS`, `COLUMN`,

Copyright © 1999, 2025, Oracle and/or its affiliates.

`MATERIALIZED VIEW`, `OPERATOR`, `TABLE`, and `VIEW`. The enhanced metadata collection enables the generation of reports useful for understanding the applications. PL/Scope can now be used as a tool to estimate the complexity of PL/SQL applications coding projects with a finer granularity than previously possible.

To collect PL/Scope data for all SQL statements used in PL/SQL source program, set the PL/SQL compilation parameter `PLSCOPE_SETTINGS` to `'STATEMENTS:ALL'`. The default value is `NONE`.

> **Note**
>
> Collecting all identifiers and statements might generate large amounts of data and slow compile time.

PL/Scope stores the data that it collects in the `SYSAUX` tablespace. If the PL/Scope collection is enabled and `SYSAUX` tablespace is unavailable during compilation of a program unit, 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

15.4 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 15-1 to display the amount of space that PL/Scope data is using.

Example 15-1 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
------------------
              1920

1 row selected.
```

> **See Also**
>
> *Oracle Database Administrator's Guide* for information about managing the `SYSAUX` tablespace

Copyright © 1999, 2025, Oracle and/or its affiliates.

15.5 Viewing PL/Scope Data

To view the data that PL/Scope has collected, you can use either:

- Static Data Dictionary Views for PL/SQL and SQL Identifiers
- Static Data Dictionary Views for SQL Statements
- SQL Developer

15.5.1 Static Data Dictionary Views for PL/SQL and SQL Identifiers

The `DBA_IDENTIFIERS` static data dictionary view family display information about PL/Scope identifiers, including their types and usages.

Topics:

- PL/SQL and SQL Identifier Types that PL/Scope Collects
- About Identifiers Usages
- Identifiers Usage Unique Keys
- About Identifiers Usage Context
- About Identifiers Signature

> **See Also**
>
> *Oracle Database Reference* for more information about the dictionary view `DBA_IDENTIFIERS` view

15.5.1.1 PL/SQL and SQL Identifier Types that PL/Scope Collects

Table 15-1 shows the identifier types that PL/Scope collects, in alphabetical order. The identifier types in Table 15-1 appear in the `TYPE` column of the `DBA_IDENTIFIERS` static data dictionary views family.

> **Note**
>
> Identifiers declared in compilation units that were not compiled with `PLSCOPE_SETTINGS='IDENTIFIERS:ALL'` do not appear in `DBA_IDENTIFIERS` static data dictionary views family.
>
> Pseudocolumns, such as ROWNUM, are not supported since they are not user defined identifiers.
>
> PL/Scope ignores column names that are literal strings.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 15-1 Identifier Types that PL/Scope Collects

TYPE Column Value	Comment
ALIAS	New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).
ASSOCIATIVE ARRAY	
COLUMN	New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).
CONSTANT	
CURSOR	
BFILE DATATYPE	Each DATATYPE is a base type declared in package STANDARD.
BLOB DATATYPE	
BOOLEAN DATATYPE	
CHARACTER DATATYPE	
CLOB DATATYPE	
DATE DATATYPE	
INTERVAL DATATYPE	
NUMBER DATATYPE	
TIME DATATYPE	
TIMESTAMP DATATYPE	
EXCEPTION	
FORMAL IN	
FORMAL IN OUT	
FORMAL OUT	
FUNCTION	
INDEX TABLE	
ITERATOR	An iterator is the index of a FOR loop.
LABEL	A label declaration also acts as a context.
LIBRARY	
MATERIALIZED VIEW	New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).
NESTED TABLE	
OBJECT	
OPAQUE	Examples of internal opaque types are ANYDATA and XMLType.
OPERATOR	New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).
PACKAGE	
PROCEDURE	
RECORD	
REFCURSOR	
SEQUENCE	New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).
SUBTYPE	

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 15-1 (Cont.) Identifier Types that PL/Scope Collects

TYPE Column Value	Comment
SYNONYM	PL/Scope does not resolve the base object name of a synonym. To find the base object name of a synonym, query *_SYNONYMS.
TABLE	New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).
TRIGGER	
UROWID	
VARIABLE	Can be object attribute, local variable, package variable, or record field.
VARRAY	
VIEW	This type is used for editioning views. New SQL identifier introduced in Oracle Database 12*c* Release 2 (12.2).

> **See Also**
>
> *Oracle Database Reference* for more information about identifiers in the stored objects

15.5.1.2 About Identifiers Usages

PL/Scope usages are the verbs that describe actions performed on behalf of the identifier.

Table 15-2 shows the usages that PL/Scope reports, in alphabetical order. The identifier actions in Table 15-2 appear in the USAGE column of the DBA_IDENTIFIERS static data dictionary views family, which is described in *Oracle Database Reference*.

Table 15-2 Usages that PL/Scope Reports

USAGE Column Value	Description
ASSIGNMENT	An assignment can be made only to an identifier that can have a value, such as a VARIABLE. Examples of assignments are: • Using an identifier to the left of an assignment operator • Using an identifier in the INTO clause of a FETCH statement • Passing an identifier to a subprogram by reference (OUT mode) • Using an identifier as the bind 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). Expressions and nested subqueries are not supported as assignment sources.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 15-2 (Cont.) Usages that PL/Scope Reports

USAGE Column Value	Description
CALL	In the context of PL/Scope, a CALL is an operation that pushes a call onto the call stack; that is: • A call to a FUNCTION or PROCEDURE • Running or fetching a cursor identifier (a logical call to SQL) A GOTO statement, or a 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. For a loop index declaration, LINE and COL (in *_IDENTIFIERS views) are the line and column of the FOR clause that implicitly declares the loop index. For a label declaration, LINE and COL are the line and column on which the label appears (and is implicitly declared) within the delimiters << and >>.
DEFINITION	A DEFINITION tells the compiler how to implement or use a previously declared identifier. Each of these types of identifiers has a DEFINITION: • EXCEPTION (can have multiple definitions) • FUNCTION • OBJECT • PACKAGE • PROCEDURE • TRIGGER For a top-level identifier only, the DEFINITION and DECLARATION are in the same place.
REFERENCE	A REFERENCE uses an identifier without changing its value. Examples of references are: • Raising an exception identifier • Using a type identifier in the declaration of a variable or formal parameter • Using a variable identifier whose type contains fields to access a field. For example, in myrecordvar.myfield := 1, a reference is made to myrecordvar, and an assignment is made to myfield. • Using a cursor for any purpose except FETCH • Passing an identifier to a subprogram by value (IN mode) • Using an identifier as the bind variable in the USING clause of an EXECUTE IMMEDIATE statement in either IN or IN OUT mode An identifier that is passed to a subprogram in IN OUT mode has both a REFERENCE usage (corresponding to IN) and an ASSIGNMENT usage (corresponding to OUT).

15.5.1.3 Identifiers Usage Unique Keys

Every identifier usage is given a numeric ID that is unique within the code unit. This is the USAGE_ID of the identifier.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- USAGE_ID

> **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).

This example shows the USAGE_ID generated for PROCEDURE p1.

```
CREATE OR REPLACE PROCEDURE p1 (a OUT VARCHAR2)
IS
  b VARCHAR2(100) := 'hello FROM p1';
BEGIN
  a := b;
END;
/

SELECT USAGE_ID, USAGE, NAME
FROM ALL_IDENTIFIERS
WHERE OBJECT_NAME = 'P1'
ORDER BY USAGE_ID;
```

```
         1 DECLARATION P1
         2 DEFINITION  P1
         3 DECLARATION A
         4 REFERENCE   VARCHAR2
         5 DECLARATION B
         6 REFERENCE   VARCHAR2
         7 ASSIGNMENT  B
         8 ASSIGNMENT  A
         9 REFERENCE   B
```

> **See Also**
>
> About Identifiers Usages for the usages in the *_IDENTIFIERS views

15.5.1.4 About Identifiers Usage Context

Identifier usages can be contexts for other identifier usages. This creates a one to many parent-child relationship among the usages. The USAGE_ID of the parent context usage is the USAGE_CONTEXT_ID for the child usages.

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.

The default top-level context, which contains all top level objects, is identified by a USAGE_CONTEXT_ID of 0.

For example:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 15-2 shows.

Example 15-2 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'
)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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
```

15.5.1.5 About Identifiers 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 15-3, which has two identifiers named `p5`, the static data dictionary view `USER_IDENTIFIERS` has several rows in which `NAME` is `p5`, but in these rows, `SIGNATURE` varies. The rows associated with the outer procedure `p5` have one signature, and the rows associated with the inner procedure `p5` have another signature. If program unit `q` calls procedure `p5`, the `USER_IDENTIFIERS` view for `q` has a row in which `NAME` is `p5` and `SIGNATURE` is the signature of the outer procedure `p5`.

Example 15-3 Program Unit with Two Identifiers Named p5

This example shows a program unit with two identifiers named p5 to demonstrate the uniqueness of the signature.

```
CREATE OR REPLACE PROCEDURE p5 AUTHID DEFINER IS
  PROCEDURE p5 IS
  BEGIN
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    DBMS_OUTPUT.PUT_LINE('Inner p5');
  END p5;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Outer p5');
  p5();
END p5;
/

SELECT LINE || ' > ' || TEXT
FROM   ALL_SOURCE
WHERE  NAME = 'P5'
   AND TYPE = 'PROCEDURE'
ORDER BY LINE;


1 > PROCEDURE p5 AUTHID DEFINER IS
2 >   PROCEDURE p5 IS
3 >   BEGIN
4 >     DBMS_OUTPUT.PUT_LINE('Inner p5');
5 >   END p5;
6 > BEGIN
7 >   DBMS_OUTPUT.PUT_LINE('Outer p5');
8 >   p5();
9 > END p5;
```

The following query shows the SIGNATURE for the PL/SQL unit is the same for its DECLARATION and DEFINITION of the inner and outer p5.

```
SELECT SIGNATURE, USAGE, LINE, COL, USAGE_ID, USAGE_CONTEXT_ID
FROM   ALL_IDENTIFIERS
WHERE OBJECT_NAME = 'P5'
ORDER BY LINE, COL, USAGE_ID;


75CD5986BA2EE5C61ACEED8C7162528F DECLARATION         1        11         1       0
75CD5986BA2EE5C61ACEED8C7162528F DEFINITION          1        11         2       1
33FB9F948F526C4B0634C0F35DFA91F6 DECLARATION         2        13         3       2
33FB9F948F526C4B0634C0F35DFA91F6 DEFINITION          2        13         4       3
33FB9F948F526C4B0634C0F35DFA91F6 CALL                8         3         7       2

CREATE OR REPLACE PROCEDURE q AUTHID DEFINER IS
BEGIN
  p5();
END q;
/


EXEC q;
Outer p5
Inner p5


SELECT SIGNATURE, USAGE, LINE, COL, USAGE_ID, USAGE_CONTEXT_ID
FROM   ALL_IDENTIFIERS
WHERE OBJECT_NAME = 'Q'
AND NAME = 'P5'
ORDER BY LINE, COL, USAGE_ID;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
75CD5986BA2EE5C61ACEED8C7162528F CALL                3          3          3       2
```

Example 15-4 Find All Usages of VARCHAR2

Identifier signatures are globally unique. This is useful to find all usages of an identifier in all units in the database. This example shows a query to find all usages of VARCHAR2.

```
SELECT UNIQUE OBJECT_NAME uses_varchar2
FROM   ALL_IDENTIFIERS
WHERE  SIGNATURE = (SELECT SIGNATURE
                    FROM   ALL_IDENTIFIERS
                    WHERE  OBJECT_NAME = 'STANDARD'
                           AND OWNER = 'SYS'
                           AND USAGE = 'DECLARATION'
                           AND NAME = 'VARCHAR2')
ORDER  BY OBJECT_NAME;
```

15.5.2 Static Data Dictionary Views for SQL Statements

The DBA_STATEMENTS static dictionary views family describes the SQL statements collected by PL/Scope.

Starting with Oracle Database 12*c* Release 2 (12.2.0.1), a new view, DBA_STATEMENTS, reports on the occurrences of static SQL in PL/SQL units. It provides information about the SQL_ID, the canonical statement text, the statement type, useful statement usage attributes, its signature, and location in the PL/SQL code. Each row represents a SQL statement instance in the PL/SQL code.

Topics:

- SQL Statement Types that PL/Scope Collects
- Statements Location Unique Keys
- About SQL Statement Usage Context
- About SQL Statements Signature

> **See Also**
>
> *Oracle Database Reference* for more information about the DBA_STATEMENTS view

15.5.2.1 SQL Statement Types that PL/Scope Collects

PL/Scope statement types represent the SQL statements used in PL/SQL.

SQL Statement types correspond to statements that can be used in PL/SQL to execute or otherwise interact with SQL. The statement type appear in the TYPE column of the DBA_STATEMENTS static data dictionary views family.

You must compile the PL/SQL units with the PLSCOPE_SETTINGS='STATEMENTS:ALL' to collect this metadata.

SQL statement types that PL/Scope collects:

- SELECT
- UPDATE

Copyright © 1999, 2025, Oracle and/or its affiliates.

- INSERT
- DELETE
- MERGE
- EXECUTE IMMEDIATE
- SET TRANSACTION
- LOCK TABLE
- COMMIT
- SAVEPOINT
- ROLLBACK
- OPEN
- CLOSE
- FETCH

15.5.2.2 Statements Location Unique Keys

Each row in the DBA_STATEMENTS view represents a unique location of a SQL statement in the PL/SQL unit. This is equivalent to unique keys within a compilation unit.

These following columns are used to determine the location of a statement in the PL/SQL code:

- `OWNER, OBJECT_NAME, OBJECT_TYPE, LINE, COL`
- `USAGE_ID`

The `USAGE_ID` is uniquely defined within a PL/SQL unit. Unlike identifiers, SQL statements do not have different usages, such as `DECLARATION`, `ASSIGNMENT`, or `REFERENCE`. All statements are considered an implicit `CALL` to the sql engine, therefore, the `DBA_STATEMENTS` view does not have the `USAGE` column, but it does use the `USAGE_ID`.

Example 15-5 Using the USAGE_ID Column to Query SQL Identifiers and Statements

```
PROCEDURE p1 (p_cust_id   NUMBER,
              p_cust_name OUT VARCHAR2)
IS
BEGIN
    SELECT (SELECT CUST_FIRST_NAME
            FROM   CUSTOMERS)
    INTO   p_cust_name
    FROM   CUSTOMERS
    WHERE  CUSTOMER_ID = p_cust_id;
END;

SELECT USAGE_ID, TYPE, NAME, USAGE, LINE, COL
FROM ( SELECT USAGE_ID, TYPE, NAME, USAGE, LINE, COL
       FROM ALL_IDENTIFIERS
       WHERE OBJECT_NAME = 'P1'
       UNION
       SELECT USAGE_ID, TYPE, 'SQL STATEMENT', " ", LINE, COL
       FROM ALL_STATEMENTS
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
       WHERE OBJECT_NAME = 'P1')
ORDER BY USAGE_ID;

USAGE_ID TYPE               NAME            USAGE        LINE   COL
---------- ----------------- --------------- ----------- ----- -----
         1 PROCEDURE         P1              DECLARATION     1    11
         2 PROCEDURE         P1              DEFINITION      1    11
         3 FORMAL IN         P_CUST_ID       DECLARATION     1    15
         4 NUMBER DATATYPE   NUMBER          REFERENCE       1    25
         5 FORMAL OUT        P_CUST_NAME     DECLARATION     1    33
         6 CHARACTER DATATYPE VARCHAR2       REFERENCE       1    49
         7 SQL STATEMENT     SELECT                          3     3
         8 TABLE             CUSTOMERS       REFERENCE       4    10
         9 FORMAL IN         P_CUST_ID       REFERENCE       4    38
        10 COLUMN            CUSTOMER_ID     REFERENCE       4    26
        11 FORMAL OUT        P_CUST_NAME     ASSIGNMENT      3    31
        12 COLUMN            CUST_FIRST_NAME REFERENCE       3    10
```

15.5.2.3 About SQL Statement Usage Context

Statements can act as a context for other statements or identifiers. Statements can also be in the context of other statements or identifiers.

The `USAGE_CONTEXT_ID` column is used to determine the context of the statement. All identifiers appearing within a statement will be in the context of that statement.

Expressions and nested subqueries are not supported as assignment sources.

Example 15-6 Using DBA_STATEMENTS USAGE_CONTEXT_ID to Query Identifiers

This example shows how to retrieve the identifiers in the context of the `SELECT` statement using the `USAGE_CONTEXT_ID` column.

```
PROCEDURE p1 (p_cust_id   NUMBER,
              p_cust_name OUT VARCHAR2)
IS
BEGIN
    SELECT (SELECT CUST_FIRST_NAME
            FROM   CUSTOMERS)
    INTO   p_cust_name
    FROM   CUSTOMERS
    WHERE  CUSTOMER_ID = p_cust_id;
END;

SELECT USAGE_ID, LPAD(' ', 2*(level-1)) || TO_CHAR(USAGE) || ' ' || NAME
usages, LINE, COL
FROM ( SELECT OBJECT_NAME, USAGE, USAGE_ID, USAGE_CONTEXT_ID, NAME, LINE, COL
       FROM ALL_IDENTIFIERS
       WHERE OBJECT_NAME = 'P1'
       UNION
       SELECT OBJECT_NAME, TYPE usage, USAGE_ID, USAGE_CONTEXT_ID,
'Statement' name, LINE, COL
       FROM ALL_STATEMENTS
       WHERE OBJECT_NAME = 'P1'
      )
      START WITH USAGE_CONTEXT_ID = 0 AND OBJECT_NAME = 'P1'
      CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID AND OBJECT_NAME = 'P1';
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  USAGE_ID USAGES                                    LINE        COL
---------- ---------------------------------- ---------- ----------
         1 DECLARATION P1                              1         11
         2   DEFINITION P1                             1         11
         3     DECLARATION P_CUST_ID                   1         15
         4       REFERENCE NUMBER                      1         27
         5     DECLARATION P_CUST_NAME                 2         33
         6       REFERENCE VARCHAR2                    2         49
         7     SELECT STATEMENT                        5          5
         8       REFERENCE CUSTOMERS                   8         12
         9       REFERENCE P_CUST_ID                   9         26
        10       REFERENCE CUSTOMER_ID                 9         12
        11       REFERENCE CUSTOMERS                   6         20
        12       REFERENCE CUST_FIRST_NAME             5         20
        13       ASSIGNMENT P_CUST_NAME                7         12
```

15.5.2.4 About SQL Statements Signature

Every SQL statement has a unique PL/Scope signature that identifies that instance of the statement in all the PL/SQL units.

The SQL statement signature distinguishes the call from a PL/SQL unit for the SQL with the same SQL_ID from another call from a different PL/SQL unit.

Nested subqueries are not individual SQL statements in ALL_STATEMENTS.

Example 15-7 Distinct SQL Signatures for the Same SQL Statement when Called from Different PL/SQL Units

This example shows two distinct signatures for the same SQL statement when it is called from PROCEDURE p1 and p2. You can observe the nested subquery is not assigned a distinct SQL_ID, therefore is not an individual SQL statements in ALL_STATEMENTS.

```
CREATE OR REPLACE PROCEDURE p1 (p_cust_id   NUMBER,
                                p_cust_name OUT VARCHAR2)
IS
BEGIN
    SELECT (SELECT CUST_FIRST_NAME
            FROM   CUSTOMERS)
    INTO   p_cust_name
    FROM   CUSTOMERS
    WHERE  CUSTOMER_ID = p_cust_id;
END;
/
CREATE OR REPLACE PROCEDURE P2 (p_cust_id   NUMBER,
                                p_cust_name OUT VARCHAR2)
IS
BEGIN
    SELECT (SELECT CUST_FIRST_NAME
            FROM   CUSTOMERS)
    INTO   p_cust_name
    FROM   CUSTOMERS
    WHERE  CUSTOMER_ID = p_cust_id;
END;
/

ACCEPT nam CHAR PROMPT "Enter OBJECT_NAME : "
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
SELECT *
FROM   ALL_STATEMENTS
WHERE OBJECT_NAME = '&&nam'
ORDER BY LINE, COL;
```

Select `ALL_STATEMENTS` for P1 and P2 to observe the different SQL signatures for the same `SQL_ID`.

```
new   3: WHERE OBJECT_NAME = 'P1'
OE
138835D3A2EBBA76A7A064E4DC14B466 SELECT
P1
PROCEDURE             7          5          5               2 c02b6yppqb46p NO  NO  NO
NO  NO  NO
YES SELECT (SELECT CUST_FIRST_NAME FROM CUSTOMERS) FROM CUSTOMERS WHERE CUSTOMER_ID
= :B1
SELECT (SELECT CUST_FIRST_NAME FROM CUSTOMERS) FROM CUSTOMERS WHERE CUSTOMER_ID      0


new   3: WHERE OBJECT_NAME = 'P2'
OE
E6A5E27E5E90997A169C5C25393FAB35 SELECT
P2
PROCEDURE             7          5          5               2 c02b6yppqb46p NO  NO  NO
NO  NO  NO
YES SELECT (SELECT CUST_FIRST_NAME FROM CUSTOMERS) FROM CUSTOMERS WHERE CUSTOMER_ID
= :B1
SELECT (SELECT CUST_FIRST_NAME FROM CUSTOMERS) FROM CUSTOMERS WHERE CUSTOMER_ID      0
```

15.5.3 SQL Developer

PL/Scope is a feature of SQL Developer. For information about using PL/Scope from SQL Developer, see SQL Developer online help

> **See Also**
>
> *Oracle SQL Developer User's Guide*

15.6 Overview of Data Dictionary Views Useful to Manage PL/SQL Code

In addition to the PL/Scope data dictionary views, the following static dictionary views are most useful for PL/SQL programmers and are most often referenced in queries related to PL/SQL code management reports. This is not an exhaustive list of all static data dictionary views.

Summary of the Data Dictionary Views Useful to Manage PL/SQL Code

View Name	Description
ALL_ARGUMENTS	Lists the arguments of the functions and procedures that are accessible to the current user

Copyright © 1999, 2025, Oracle and/or its affiliates.

View Name	Description
ALL_DEPENDENCIES	Describes dependencies between procedures, packages, functions, package bodies, and triggers accessible to the current user
ALL_ERRORS	Describes the current errors on the stored objects accessible to the current user
ALL_IDENTIFIERS	Displays information about the identifiers in the stored objects accessible to the current user
USER_OBJECT_SIZE	Describes the size, in bytes, of PL/SQL objects owned by the current user. Although this information is meant to be used by the compiler and runtime engine, you can use it to identify the large programs in your environment.
ALL_OBJECTS	Describes all objects accessible to the current user
ALL_PLSQL_COLL_TYPES	Describes the named PL/SQL collection types accessible to the user
ALL_PLSQL_OBJECT_SETTINGS	Displays information about the compiler settings for the stored objects accessible to the current user
ALL_PLSQL_TYPES	Describes the PL/SQL types accessible to the user
ALL_PLSQL_TYPE_ATTRS	Describes the attributes of PL/SQL types accessible to the user
ALL_PROCEDURES	Describes all PL/SQL functions and procedures, along with associated properties, that are accessible to the current user
ALL_SEQUENCES	Describes the sequences accessible to the current user
ALL_SOURCE	Describes the text source of the stored objects accessible to the current user
ALL_STATEMENTS	Describes all SQL statements in stored PL/SQL objects accessible to the user
ALL_STORED_SETTINGS	Describes the persistent parameter settings for stored PL/SQL units for which the current user has execute privileges
ALL_SYNONYMS	Describes the synonyms accessible to the current user
ALL_TAB_COLUMNS	Describes the columns of the tables, views, and clusters accessible to the current user
ALL_TABLES	Describes the relational tables accessible to the current user
ALL_TRIGGERS	Describes the triggers on tables accessible to the current user
ALL_VIEWS	Describes the views accessible to the current user

15.7 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

16

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

16.1 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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

* 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 or raw profiler data table.

- Analyzer

  The analyzer component processes the raw profiler output and produce analyzed results. The analyzer component analyzes:

  – Raw profiler data located in the raw profiler data file and raw profiler data table into HTML CLOB report, analyzed report file, and hierarchical profiler analysis tables.

> **Note**
>
> To generate simple HTML reports directly from raw profiler output, without using the Analyzer, you can use the `plshprof` command-line utility.

16.2 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.

Example 16-1 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
-- Run procedure to be profiled
BEGIN
  test;
END;
/
BEGIN
  -- Stop profiling
  DBMS_HPROF.STOP_PROFILING;
END;
/
```

Example 16-2 Profiling with Raw Profiler Data Table

```
DECLARE
analyze_runid number;
trace_id number;
BEGIN
-- create raw profiler data and analysis tables
-- call create_tables with force_it =>FALSE ( default) when
-- raw profiler data and analysis tables do not exist already
DBMS_HPROF . CREATE_TABLES ;
-- Start profiling .
-- Write raw profiler data in raw profiler data table
trace_id := DBMS_HPROF . START_PROFILING ;
-- Run procedure to be profiled
test ;
-- Stop profiling
DBMS_HPROF . STOP_PROFILING ;
-- analyzes trace_id entry in raw profiler data table and writes
-- hierarchical profiler information in hprof 's analysis tables.
analyze_runid := DBMS_HPROF . ANALYZE(trace_id );
END;
/
```

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;
/
```

Consider the PL/SQL procedure that analyzes and generates HTML CLOB report from raw profiler data table

```
declare
reportclob clob ;
trace_id number;
begin
-- create raw profiler data and analysis tables
-- force_it =>TRUE will dropped the tables if table existed
DBMS_HPROF . CREATE_TABLES (force_it =>TRUE );
-- Start profiling .
-- Write raw profiler data in raw profiler data table
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
trace_id := DBMS_HPROF . START_PROFILING ;
-- Run procedure to be profiled
test ;
-- Stop profiling
DBMS_HPROF . STOP_PROFILING ;
-- analyzes trace_id entry in raw profiler data table and produce
-- analyzed HTML report in reportclob .
DBMS_HPROF .ANALYZE (trace_id , reportclob );
end;
/
```

The SQL script in Example 16-1 profiles the execution of the PL/SQL procedure `test`.

> **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 16-1, 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;
> ```

> **See Also**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about using directory objects
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_HPROF.START_PROFILING` and `DBMS_HPROF.STOP_PROFILING`

16.3 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 16-1 wrote this raw profiler output to the file `test.trc`:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
P#V PLSHPROF Internal Version 1.0
P#! PL/SQL Timer Started
P#C PLSQL."".""."__plsql_vm"
P#X 4
P#C PLSQL."".""."__anonymous_block"
P#X 77
P#C PLSQL."HR"."TEST"::7."TEST"#980980e97e42f8ec #1
P#X 4
P#C PLSQL."HR"."TEST"::7."TEST.FOO"#980980e97e42f8ec #4
P#X 47
P#C SQL."HR"."TEST"::7."__static_sql_exec_line6" #6."3r6qf2qhr3cm1"
P#! SELECT COUNT(*) FROM EMPLOYEES
P#X 279
P#R
P#X 58
P#R
P#X 3
P#C PLSQL."HR"."TEST"::7."TEST.FOO"#980980e97e42f8ec #4
P#X 4
P#C SQL."HR"."TEST"::7."__static_sql_exec_line6" #6."3r6qf2qhr3cm1"
P#! SELECT COUNT(*) FROM EMPLOYEES
P#X 121
P#R
P#X 5
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."3r6qf2qhr3cm1"
P#! SELECT COUNT(*) FROM EMPLOYEES
P#X 117
P#R
P#X 4
P#R
P#X 2
P#R
P#X 2
P#R
P#X 3
P#R
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
P#X 86
P#C PLSQL."SYS"."DBMS_HPROF"::11."STOP_PROFILING"#980980e97e42f8ec #453
P#R
P#R
P#R
P#! PL/SQL Timer Stopped
```

Table 16-1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 16-1 (Cont.) Raw Profiler Output File Indicators

Indicator	Meaning
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
- **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 Special Function Names.
- Hexadecimal value that represents the hash algorithm of the **signature** of the called subprogram

  The DBMS_HPROF.analyze PL/SQL API 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 the hash algorithm 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> When a subprogram is inlined, it is not reported in the profiler output.
>
> When a call to a `DETERMINISTIC` function is "optimized away," it is not reported in the profiler output.

> **See Also**
>
> - Namespaces of Tracked Subprograms
> - Analyzing Profile Data for more information about `analyze` PL/SQL API
> - *Oracle Database PL/SQL Language Reference* for information about subprogram inlining
> - *Oracle Database PL/SQL Language Reference* for information about `DETERMINISTIC` functions

16.3.1 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

16.3.2 Special Function Names

PL/SQL hierarchical profiler tracks certain operations as if they were functions with the names and namespaces shown in Table 16-2.

Table 16-2 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 *line#*	`__static_sql_exec_line`*line#*	SQL
Dynamic SQL statement at line *line#*	`__dyn_sql_exec_line`*line#*	SQL

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 16-2 (Cont.) Function Names of Operations that the PL/SQL Hierarchical Profiler Tracks

Tracked Operation	Function Name	Namespace
SQL FETCH statement at line *line#*	`__sql_fetch_line`*line#*	SQL

16.4 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 16-3.

Table 16-3 PL/SQL Hierarchical Profiler Database Tables

Table	Description
`DBMSHP_RUNS`	Top-level information for this run of `DBMS_HPROF.analyze`. For column descriptions, see Table 16-4.
`DBMSHP_FUNCTION_INFO`	Information for each subprogram profiled in this run of `DBMS_HPROF.analyze`. For column descriptions, see Table 16-5.
`DBMSHP_PARENT_CHILD_INFO`	Parent-child information for each subprogram profiled in this run of `DBMS_HPROF.analyze`. For column descriptions, see Table 16-6.

Topics:

- Creating Hierarchical Profiler Tables
- Understanding Hierarchical Profiler Tables

> **Note**
>
> To generate simple HTML reports directly from raw profiler output, without using the Analyzer, you can use the `plshprof` command-line utility. For details, see plshprof Utility.

16.4.1 Creating Hierarchical Profiler Tables

The following steps explain how to create hierarchical profiler tables in Table 16-3 and the other data structures required for persistently storing profile data, privileges required to run the `DBMS_HPROF` package, and generate custom reports:

1. Hierarchical profiler tables in Table 16-3 and other data structures required for persistently storing profile data can be created in the following ways.
   a. Call the `DBMS_HPROF.CREATE_TABLES` procedure.
   b. Run the script `dbmshptab.sql` (located in the directory `rdbms/admin`).

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> Running the script `dbmshptab.sql` drops any previously created hierarchical profiler tables.

> **Note**
>
> The `dbmshptab.sql` (located in the directory `rdbms/admin`) has been deprecated. This script contains the statements to drop the tables and sequences along with the deprecation notes.

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 Understanding Raw Profiler Output.)

   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.

Example 16-3 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;
/
```

The anonymous block in Example 16-3:

- Invokes the function `DBMS_HPROF.analyze` function, which:
  - Analyzes the profile data in the raw profiler output file `test.trc` (from Understanding Raw Profiler Output), 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 16-3.
  - Returns a unique identifier that you can use to query the hierarchical profiler tables in Table 16-3. (By querying these hierarchical profiler tables, you can produce customized reports.)
- Stores the unique identifier in the variable `runid`, which it prints.

16.4.2 Understanding Hierarchical Profiler Tables

These topics explain how to use the hierarchical profiler tables in Table 16-3:

- Hierarchical Profiler Database Table Columns

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Distinguishing Between Overloaded Subprograms
- Hierarchical Profiler Tables for Sample PL/SQL Procedure
- Examples of Calls to DBMS_HPROF.analyze with Options

16.4.2.1 Hierarchical Profiler Database Table Columns

Table 16-4 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 16-4 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 16-5 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 16-5 has a row for each variation of that subprogram. Each variation has its own LINE# and HASH (see Distinguishing Between Overloaded Subprograms.)

The primary key for the hierarchical profiler table DBMSHP_FUNCTION_INFO is RUNID, SYMBOLID.

Table 16-5 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 16-4.
SYMBOLID	NUMBER	Symbol identifier for subprogram (unique for this run of DBMS_HPROF.analyze).
OWNER	VARCHAR2(128)	Owner of module in which each subprogram is defined (for example, SYS or HR).
MODULE	VARCHAR2(128)	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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 16-5 (Cont.) DBMSHP_FUNCTION_INFO Table Columns

Column Name	Column Data Type	Column Contents
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 Special Function Names. For subprogram B defined within subprogram A, this name is A.B. A recursive call to function X is denoted X@*n*, where *n* is the recursion depth. For example, X@1 is the first recursive call to X.
LINE#	NUMBER	Line number in OWNER.MODULE at which FUNCTION is defined.
HASH	RAW(32)	Hash code for signature of subprogram (unique for this run of DBMS_HPROF.analyze).
NAMESPACE	VARCHAR2(32)	Namespace of subprogram. For details, see Namespaces of Tracked Subprograms.
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.
SQL_ID	VARCHAR2(13)	SQL Identifier of the SQL statement.
SQL_TEXT	VARCHAR2(4000)	First 50 characters of the SQL statement.

Table 16-6 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 16-6 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 16-5.
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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 16-6 (Cont.) DBMSHP_PARENT_CHILD_INFO Table Columns

Column Name	Column Data Type	Column Contents
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.

16.4.2.2 Distinguishing Between Overloaded Subprograms

Overloaded subprograms are different subprograms with the same name.

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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for more information about overloaded subprograms

16.4.2.3 Hierarchical Profiler Tables for Sample PL/SQL Procedure

The hierarchical profiler tables for the PL/SQL procedure `test` in Collecting Profile Data are shown in Example 16-4 through Example 16-6.

Consider the third row of the table `DBMSHP_PARENT_CHILD_INFO` (Example 16-6). 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 16-5) 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 16-4 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 16-5 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                                      __anonymous_block
         3           2                                      __plsql_vm
         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 16-6 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      1
         3          1          5                     77                       61      1
         3          2          1                    121                       44      1
         3          3          4                      1                        1      1
         3          5          3                      9                        8      1
         3          5          4                      7                        7      1
```

16.4.2.4 Examples of Calls to DBMS_HPROF.analyze with Options

For an example of a call to `DBMS_HPROF.analyze` without options, see Example 16-3.

Example 16-7 creates a package, creates a procedure that invokes subprograms in the package, profiles the procedure, and uses `DBMS_HRPROF.analyze` to analyze the raw profiler output file. The raw profiler output file is in the directory corresponding to the `PLSHPROF_DIR` directory object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 16-7 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;
/
-- If not done, create hierarchical profiler tables (see Creating Hierarchical Profiler Tables).

-- Call DBMS_HPROF.analyze with options

DECLARE
  runid NUMBER;
BEGIN
  -- Analyze only subtrees rooted at trace entry "HR"."PKG"."MYPROC"

  runid := DBMS_HPROF.analyze('PLSHPROF_DIR', 'test2.trc',
                              trace => '"HR"."PKG"."MYPROC"');

  -- 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;
/
```

16.5 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 Collecting Profile Data.)

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

16.5.1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

Option	Description	Default
`-skip count`	Skips first `count` calls. Use only with `-trace symbol`.	0
`-collect count`	Collects information for `count` calls. Use only with `-trace symbol`.	1
`-output filename`	Specifies name of output file	`symbol.html` or `tracefile1.html`
`-summary`	Prints only elapsed time	None
`-trace symbol`	Specifies function name of tree root	Not applicable

Suppose that your raw profiler output file, `test.trc`, is in the current directory. You want to analyze and generate HTML reports, and you want the root file of the HTML report to be named `report.html`. Use this command (`%` is the prompt):

```
% plshprof -output report test.trc
```

16.5.2 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
- Understanding PL/SQL Hierarchical Profiler SQL-Level Reports
- Module-Level Reports
- Namespace-Level Reports
- Parents and Children Report for a Function

16.5.2.1 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

824 microsecs (elapsed time) & 12 function calls

Copyright © 1999, 2025, Oracle and/or its affiliates.

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)
- SQL ID Elapsed Time (microsecs) Data sorted by SQL ID

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Descendants Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Mean Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Function Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Descendants Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Module Name
- Module Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- Namespace Elapsed Time (microsecs) Data sorted by Namespace
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Parents and Children Elapsed Time (microsecs) Data

16.5.2.2 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.
- SQL ID
- SQL Text (First 50 characters of the SQL text).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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)

824 microsecs (elapsed time) & 12 function calls

Subtree	Ind %	Function	Descendant	Ind %	Calls	Ind %	Function Name	SQL ID	SQL TEXT
824	**100 %**	10	814	98.8 %	2	16.7 %	__plsq_vm		
814	**98.8 %**	165	649	78.8 %	2	16.7 %	__anonymous_block		
649	**78.8 %**	11	638	77.4 %	1	8.3 %	HR.TEST.TEST (Line 1)		
638	**77.4 %**	121	517	62.7 %	3	25.0 %	HR.TEST.TEST.FOO (Line 4)		
517	**62.7 %**	517	0	0.0 %	3	25.0 %	HR.TEST.__static_sql_exec_line5 (Line 6)	3r6qf2qhr3cm1	SELECT COUNT(*) FROM EMPLOYEES
0	**0.0 %**	0	0	0.0 %	1	8.3 %	SYS.DBMS_HPROF.STOP_PROFILING (Line 453)		

16.5.2.3 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

16.5.2.4 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

16.5.2.5 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.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 microsecs (8.0%).

Sample Report

HR.P.UPLOAD (Line 3)

Subtree	Ind%	Function	Ind%	Descendant	Ind%	Calls	Ind%	Function Name
166860	**100%**	11713	7.0%	155147	93.0%	2	0.2%	HR.P.UPLOAD (Line 3)
Parents:								
106325	**63.7%**	6434	54.9%	99891	64.4%	1	50.0%	HR.UTILS.COPY_IMAGE (Line 3)
60535	**36.3%**	5279	45.1%	55256	35.6%	1	50.0%	HR.UTILS.COPY_FILE (Line 8))
Children:								
71818	**46.3%**	71818	100%	0	N/A	2	100%	HR.P.__static_sql_exec_line38 (Line 38)
67649	**43.6%**	67649	100%	0	N/A	214	100%	SYS.DBMS_LOB.WRITEAPPEND (Line 926)
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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Subtree	Ind%	Function	Ind%	Descendant	Ind%	Calls	Ind%	Function Name
100	**0.1%**	100	100%	0	N/A	2	100%	SYS.DBMS_LOB.CREATETEMPORARY (Line 536)

> **See Also**
>
> Function-Level Reports

16.5.2.6 Understanding PL/SQL Hierarchical Profiler SQL-Level Reports

Understanding DBMS_HPROF.ANALYZE SQL-level reports.

The PL/SQL Hierarchical Profiler SQL-level report provides the list of all the SQLs collected during profiling. along with the abbreviated SQL text, and the elapsed time (microsecs) sorted by SQL ID. The SQL ID is useful if other SQL statistics must be retrieved in other tables, for example, for SQL tuning purpose. The first 50 characters of the SQL text is included in the report. You can use the Function-Level reports to get the details surrounding where the SQL is called, and its location in the source code if needed.

Sample Report

SQL ID Elapsed Time (microsecs) Data sorted by SQL ID

824 microsecs (elapsed time) & 12 function calls

SQL ID	SQL TEXT	Function	Ind%	Calls	Ind%
3r6qf2qhr3cm1	SELECT COUNT(*) FROM EMPLOYEES	679	82.4%	3	25.0%

16.5.3 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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

16.5.3.1 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.

16.5.3.2 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)#
-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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

16.5.3.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Sample Report 1

Function Elapsed Time (microsecs) Data for Performance Regressions

Subtree	Function	Rel%	Ind%	Cum%	Descendant	Calls	Rel%	Mean Function	Rel%	Function Name
4075787	**2075627**	**+941%**	**61.1%**	61.1%	2000160	0		2075627	+941%	HR.P.G (Line 35)
1101384	**1101384**	**+54.6%**	**32.4%**	93.5%	0	5	+55.6%	-1346	-0.6%	HR.P.H (Line 18)
222371	**222371**		6.5%	100%	0	1				HR.P.J (Line 10)#

The report in Sample Report 2 shows the difference information for all functions that performed better in the second run than they did in the first run.

Sample Report 2

Function Elapsed Time (microsecs) Data for Performance Improvements

Subtree	Function	Rel%	Ind%	Cum%	Descendant	Calls	Rel%	Mean Function	Rel%	Function Name
-1365827	**-467051**	**-50.0%**	**67.7%**	67.7%	-898776	-2	-50.0%	-32	0.0%	HR.P.F (Line 25)
-222737	**-222737**		**32.3%**	100%	0	-1				HR.P.I (Line 2)
2709589	**-5**	**-21.7%**	**0.0%**	100%	2709594	0		-5	-20.8	HR.P.TEST (Line 46)#

The report in Sample Report 3 summarizes the difference information for all functions.

Sample Report 3

Function Elapsed Time (microsecs) Data sorted by Total Function Call Count Delta

Subtree	Function	Rel%	Ind%	Descendant	Calls	Rel%	Mean Function	Rel%	Function Name
1101384	1101384	+54.6%	32.4%	0	**5**	**+55.6%**	-1346	-0.6%	HR.P.H (Line 18)
-1365827	-467051	+50.0%	67.7%	-898776	**-2**	**-50.0%**	-32	-0.0%	HR.P.F (Line 25)
-222377	-222377		32.3%	0	**-1**				HR.P.I (Line 2)#
222371	222371		6.5%	0	**1**				HR.P.J(Line 10)#
4075787	2075627	+941%	61.1%	2000160	**0**		2075627	+941%	HR.P.G (Line 35)
2709589	-5	-21.7%	0.0%	2709594	**0**		-5	-20.8%	HR.P.TEST (Line 46)
0	0			0	**0**				SYS.DBMS_HPROF.STOP_PROFILING (Line 53)

Copyright © 1999, 2025, Oracle and/or its affiliates.

16.5.3.4 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

16.5.3.5 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**

16.5.3.6 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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%

Copyright © 1999, 2025, Oracle and/or its affiliates.

17

Using PL/SQL Basic Block Coverage to Maintain Quality

The PL/SQL basic block coverage interface helps you ensure some quality, predictability and consistency, by assessing how well your tests exercise your code.

The code coverage measurement tests are typically executed on a test environment, not on a production database. The goal is to maintain or improve the regression tests suite quality over the lifecycle of multiple PL/SQL code releases. PL/SQL code coverage can help you answer questions such as:

- Is your testing suites development keeping up with the development of your new code?
- Do you need more tests?

The PL/SQL basic block coverage interface collects coverage data for PL/SQL units exercised in a test run.

Topics:

- Overview of PL/SQL Basic Block Coverage
- Collecting PL/SQL Code Coverage Data
- PL/SQL Code Coverage Tables Description

See Also

- *Oracle Database PL/SQL Language Reference* for the `COVERAGE PRAGMA` syntax and semantics
- *Oracle Database PL/SQL Packages and Types Reference* for more information about using the `DBMS_PLSQL_CODE_COVERAGE` package
- *Oracle Database PL/SQL Language Reference* for more information about the PLSQL_OPTIMIZE_LEVEL compilation parameter

17.1 Overview of PL/SQL Basic Block Coverage

The `DBMS_PLSQL_CODE_COVERAGE` package enables you to collect data at the basic block level. PL/SQL developers and testing engineers use code coverage testing results as part of their standard quality assurance metric.

Code coverage is a measure of the percentage of code which is covered by automated tests. A program with high code coverage has less chance of containing bugs than a program with low code coverage. The most important is the percent of basic blocks executed by a test suite. A basic block is a linear segment of code with no branches. A basic block has a single entry point (no code within a basic block is the destination of a jump instruction) and a single exit point (only the last instruction, or an exception, can move the point of execution to a different basic block). Basic block boundaries cannot be predicted by visual inspection of the code. The compiler generates the blocks that are executed at runtime.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Coverage information at the unit level can be derived accurately by collecting coverage at the basic block level. Utilities can be produced to report and visualize the test coverage results and help identify code that is covered, partially covered, or not covered by tests.

It is not always feasible to write test cases to exercise some basic blocks. It is possible to exclude these blocks from the coverage calculation by marking them using the `COVERAGE` pragma. The source code can be marked as not feasible for coverage either a single basic block, or a range of basic blocks.

17.2 Collecting PL/SQL Code Coverage Data

This example shows you the basic steps to collect and analyze PL/SQL basic block code coverage data using the `DBMS_PLSQL_CODE_COVERAGE` package.

PL/SQL basic block coverage data is collected when program units use `INTERPRETED` compilation (parameter set `PLSQL_CODE_TYPE = INTERPRETED`). PL/SQL basic block coverage data is not collected when program units use `NATIVE` compilation. You can disable the `NATIVE` compiler by setting the parameter `PLSQL_OPTIMIZE_LEVEL <= 1`. Regardless of the compilation mode, coverage data for wrapped units is not collected.

Follow these steps to collect and analyze PL/SQL basic block code coverage data using the `DBMS_PLSQL_CODE_COVERAGE` package.

1. Run the procedure `CREATE_COVERAGE_TABLES` to create the tables required by the package to store the coverage data. You only need to run this step once as part of setup.

   ```
   EXECUTE DBMS_PLSQL_CODE_COVERAGE.CREATE_COVERAGE_TABLES;
   ```

2. Start the coverage run.

   ```
   DECLARE    testsuite_run NUMBER;
   BEGIN
      testsuite_run :=  DBMS_PLSQL_CODE_COVERAGE.START_COVERAGE(RUN_COMMENT => 'Test
   Suite ABC');
   END;
   /
   ```

3. Run the tests.
4. Stop the coverage collection and write the data to the collection tables.

   ```
   EXECUTE DBMS_PLSQL_CODE_COVERAGE.STOP_COVERAGE;
   ```

5. Query the coverage tables to inspect results.

17.3 PL/SQL Code Coverage Tables Description

The following tables are created by the `PLSQL_CODE_COVERAGE.CREATE_COVERAGE_TABLES` procedure to collect code coverage data.

The `DBMSPCC_RUNS` table contains one row for each execution of the `DBMS_PLSQL_CODE_COVERAGE.START_COVERAGE` function. The primary key is the `RUN_ID`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 17-1 DBMSPCC_RUNS Table Columns

Column Name	Column Data Type	Description
RUN_ID	NUMBER(38)	Unique identifier automatically generated for this run of DBMS_PLSQL_CODE_COVERAGE.START_COVERAGE
RUN_COMMENT	VARCHAR2(4000)	User comment to identify the run
RUN_OWNER	VARCHAR2(128)	User who started the run
RUN_TIMESTAMP	DATE	Date timestamp when the run started

The DBMSPCC_UNITS table contains the PL/SQL units information exercised in a run. The primary key is RUN_ID, and OBJECT_ID. The OBJECT_ID and LAST_DDL_TIME allows you to determine if a unit has been modified since the run started by comparing to the object LAST_DDL_TIME in the static data dictionary view ALL_OBJECTS.

Table 17-2 DBMSPCC_UNITS Table Columns

Column Name	Column Data Type	Description
RUN_ID	NUMBER	References the RUN_ID column in the DBMSPCC_RUNS table
OBJECT_ID	NUMBER	Unique identifier for the unit
OWNER	VARCHAR2(128)	Owner of the unit
NAME	VARCHAR2(128)	Unit name
TYPE	VARCHAR2(12)	Unit type
LAST_DDL_TIME	DATE	Date timestamp for the last modification of the unit resulting from a DDL statement captured at run time

The DBMSPCC_BLOCKS table identifies all the blocks in a unit. The block location is indicated by its starting position in the source code (LINE, COL). The primary key is RUN_ID, OBJECT_ID and BLOCK. It is implicit that one block ends at the character position immediately before that of the start of the next. More than one block can start at the same location. If a unit has not been modified since the run started, the source code lines can be extracted from the static data dictionary view ALL_SOURCE.

Table 17-3 DBMSPCC_BLOCKS Table Columns

Column Name	Column Data Type	Description
RUN_ID	NUMBER(38)	References the RUN_ID column in the DBMSPCC_UNITS table
OBJECT_ID	NUMBER(38)	References the OBJECT_ID in the DBMSPCC_UNITS table
BLOCK	NUMBER(38)	Basic block number
LINE	NUMBER(38)	Starting line number of the basic block

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 17-3 (Cont.) DBMSPCC_BLOCKS Table Columns

Column Name	Column Data Type	Description
COL	NUMBER(38)	Starting column number of basic block
COVERED	NUMBER(1)	Set to 1 if a basic block is covered, or to 0 otherwise
NOT_FEASIBLE	NUMBER(1)	Set to 1 if a basic block is marked as NOT_FEASIBLE, or to 0 otherwise

Copyright © 1999, 2025, Oracle and/or its affiliates.

18

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.

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

> **See Also**
>
> Oracle Application Express App Builder User's Guide for information about using Oracle Application Express

18.1 Overview of PL/SQL Web Applications

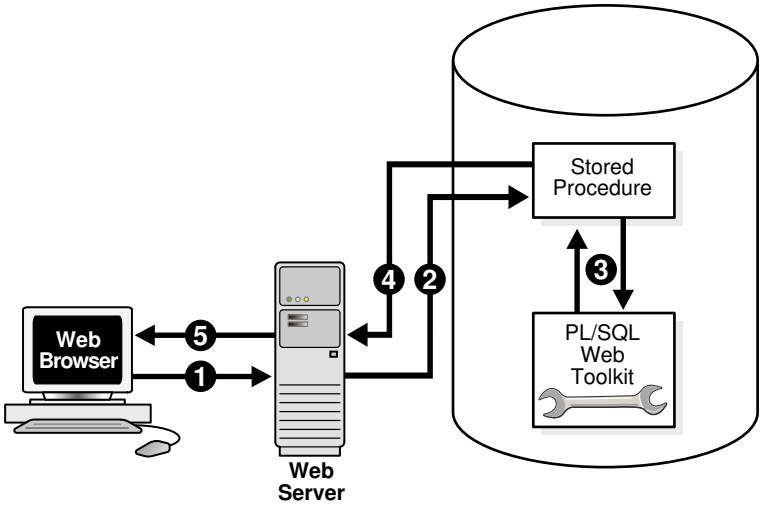
Typically, a web application written in PL/SQL is a set of stored subprograms that interact with web browsers through HTTP. A set of interlinked, dynamically generated HTML pages forms the user interface of a web application.

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 18-1 illustrates the generic process for a PL/SQL web application.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 18-1 PL/SQL Web Application



18.2 Implementing PL/SQL Web Applications

You can implement a web browser-based application entirely in PL/SQL with PL/SQL Gateway or with PL/SQL Web Toolkit.

Topics:

- PL/SQL Gateway
- PL/SQL Web Toolkit

18.2.1 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.

18.2.1.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

18.2.1.2 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.

18.2.2 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 18-1 describes commonly used PL/SQL Web Toolkit packages.

Table 18-1 Commonly Used Packages in the PL/SQL Web Toolkit

Package	Description of Contents
`HTF`	Function versions of the subprograms in the `htp` package. The function versions do not directly generate output in a web page. Instead, they pass their output as return values to the statements that invoke them. Use these functions when you must nest function calls.
`HTP`	Subprograms that generate HTML tags. For example, the procedure `htp.anchor` generates the HTML anchor tag, <A>.
`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 18-1 (Cont.) Commonly Used Packages in the PL/SQL Web Toolkit

Package	Description of Contents
OWA_IMAGE	Subprograms that obtain the coordinates where a user clicked an image. Use this package when you have an image map whose destination links invoke a PL/SQL gateway.
OWA_OPT_LOCK	Subprograms that impose database optimistic locking strategies to prevent lost updates. Lost updates can otherwise occur if a user selects, and then attempts to update, a row whose values were changed in the meantime by another user.
OWA_PATTERN	Subprograms that perform string matching and string manipulation with regular expressions.
OWA_SEC	Subprograms used by the PL/SQL gateway for authenticating requests.
OWA_TEXT	Subprograms used by package OWA_PATTERN for manipulating strings. You can also use them directly.
OWA_UTIL	These types of utility subprograms: • Dynamic SQL utilities to produce pages with dynamically generated SQL code. • HTML utilities to retrieve the values of CGI environment variables and perform URL redirects. • Date utilities for correct date-handling. Date values are simple strings in HTML, but must be properly treated as an Oracle Database data type.
WPG_DOCLOAD	Subprograms that download documents from a document repository that you define using the DAD configuration.

See Also

Oracle AI Database PL/SQL Packages and Types Reference for syntax, descriptions, and examples for the PL/SQL Web Toolkit packages

18.3 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

18.4 Using Embedded PL/SQL Gateway

The embedded gateway functions very similar to the `mod_plsql` gateway.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Topics:

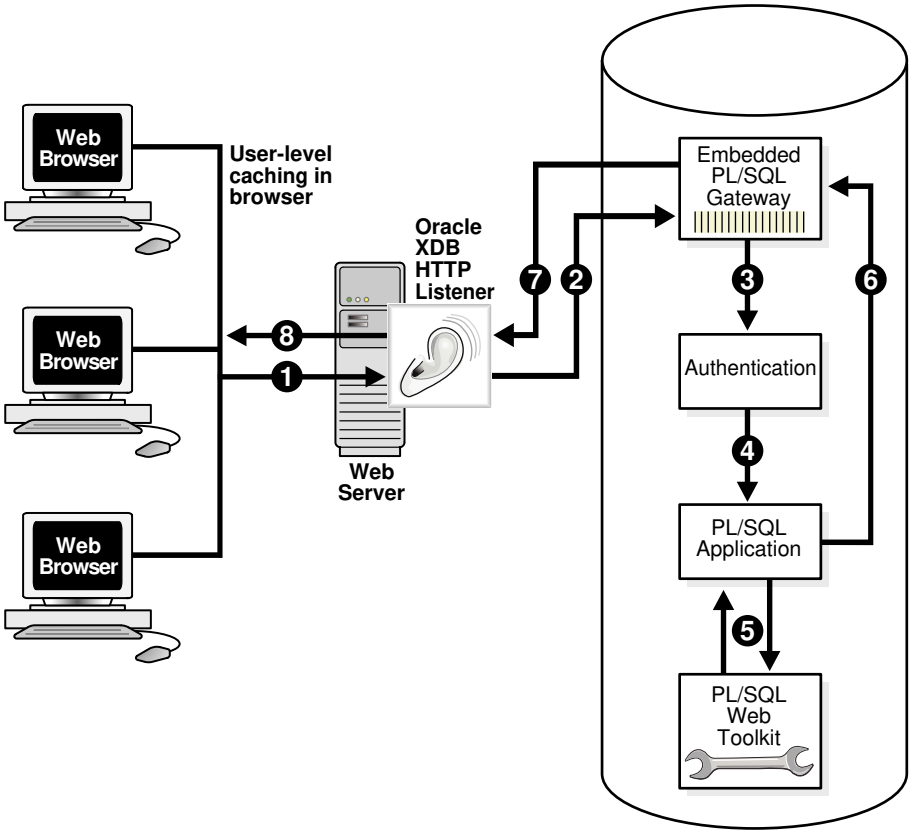
- How Embedded PL/SQL Gateway Processes Client Requests
- Installing Embedded PL/SQL Gateway
- Configuring Embedded PL/SQL Gateway
- Invoking PL/SQL Stored Subprograms Through Embedded PL/SQL Gateway
- Securing Application Access with Embedded PL/SQL Gateway
- Restrictions in Embedded PL/SQL Gateway
- Using Embedded PL/SQL Gateway: Scenario

> **See Also**
>
> *Oracle HTTP Server mod_plsql User's Guide*

18.4.1 How Embedded PL/SQL Gateway Processes Client Requests

Figure 18-2 illustrates the process by which the embedded gateway handles client HTTP requests.

Figure 18-2 Processing Client Requests with Embedded PL/SQL Gateway



Copyright © 1999, 2025, Oracle and/or its affiliates.

The explanation of the steps in Figure 18-2 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.

18.4.2 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.

The PL/SQL Web Toolkit is part of the standard installation of the database, so no supplementary installation is necessary.

> **See Also**
>
> *Oracle XML DB Developer's Guide* for information about manually adding Oracle XML DB to an existing database

Copyright © 1999, 2025, Oracle and/or its affiliates.

18.4.3 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

18.4.3.1 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 18-2 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 18-2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 18-2 (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
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
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`
- `PlsqlDatabaseConnectString` (because the embedded gateway does not support logon to external databases)

Like the DAD attributes, the global configuration parameters are optional. Table 18-3 describes the `DBMS_EPG` global attributes and the corresponding `mod_plsql` global parameters.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 18-3 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

18.4.3.2 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
- Examples: Creating and Configuring DADs
- Example: Determining the Authentication Mode for a DAD
- Example: Determining the Authentication Mode for All DADs
- Example: Showing DAD Authorizations that Are Not in Effect
- Examining Embedded PL/SQL Gateway Configuration

Copyright © 1999, 2025, Oracle and/or its affiliates.

18.4.3.2.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

18.4.3.2.2 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.

18.4.3.2.3 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/*');
   ```

3. Set the DAD attribute `database-username` to `ANONYMOUS`. For example:

   ```
   EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('HR_DAD', 'database-username', 'ANONYMOUS');
   ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

18.4.3.2.4 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 18-4 shows how the answers to the preceding questions determine the authentication mode.

Table 18-4 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

18.4.3.2.5 Examples: Creating and Configuring DADs

Example 18-1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Creates a DAD with dynamic authentication that is restricted to the HR account.

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 18-2 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 18-1 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');
```

Example 18-2 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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');
```

18.4.3.2.6 Example: Determining the Authentication Mode for a DAD

Example 18-3 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.

Assume that you have run the script in Example 18-1 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'.
```

Example 18-3 Determining the Authentication Mode for a DAD

```
CREATE OR REPLACE PROCEDURE show_dad_auth_status (p_dadname VARCHAR2) IS
  v_daduser VARCHAR2(128);
  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 ||
      ''' 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 ||
      ''' is set up for dynamic authentication for user ''' ||
      v_daduser || ''' only.');
  ELSE
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    DBMS_OUTPUT.PUT_LINE (
      '''' || p_dadname ||
      ''' is set up for dynamic authentication for any user.');
  END IF;
END;
/
```

18.4.3.2.7 Example: Determining the Authentication Mode for All DADs

The anonymous block in Example 18-4 reports the authentication modes of all registered DADs. It invokes the `show_dad_auth_status` procedure from Example 18-3.

If you have run the script in Example 18-1 to create and configure various DADs, the output of Example 18-4 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.
```

Example 18-4 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;
/
```

18.4.3.2.8 Example: Showing DAD Authorizations that Are Not in Effect

The anonymous block in Example 18-5 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

If you have run the script in Example 18-2 to create and configure various DADs, the output of Example 18-5 (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.
```

Example 18-5 Showing DAD Authorizations that Are Not in Effect

```
DECLARE
  v_dad_names DBMS_EPG.VARCHAR2_TABLE;
  v_dad_user  VARCHAR2(128);
  v_dad_found BOOLEAN;
BEGIN
  DBMS_OUTPUT.PUT_LINE
    ('---------- DAD Authorizations Not in Effect ----------');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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;
/
```

18.4.3.2.9 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 18-6 shows the output of the `epgstat.sql` script for Example 18-1 when the `ANONYMOUS` account is locked.

Example 18-6 epgstat.sql Script Output for Example 18-1

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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
+--------------------------+
| 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_Restric
ted

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                                     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.            |
+-----------------------------------------------------+
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.
```

18.4.4 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`.

See Also

- *Oracle HTTP Server mod_plsql User's Guide* for the following topics:
  - Transaction mode
  - Supported data types
  - Parameter-passing scheme
  - File upload and download support
  - Path-aliasing
  - Common Gateway Interface (CGI) environment variables

18.4.5 Securing Application Access with Embedded PL/SQL Gateway

The embedded gateway shares the same protection mechanism with `mod_plsql`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle HTTP Server mod_plsql User's Guide*

18.4.6 Restrictions in Embedded PL/SQL Gateway

The `mod_plsql` restrictions 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

> **See Also**
>
> - *Oracle HTTP Server mod_plsql User's Guide* for more information about restrictions
> - *Oracle HTTP Server mod_plsql User's Guide* for information about authentication modes

18.4.7 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');
   ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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 they 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

18.5 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 18-7 shows how to generate a simple HTML page by calling the `HTP` functions that correspond to each HTML tag.

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 18-8 illustrates this technique.

Example 18-7 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;
/
```

Example 18-8 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>');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
END;
/
```

18.6 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

18.6.1 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 18-9 shows a simple drop-down list.

Example 18-9 HTML Drop-Down List

```
<form>
<select name="seasons">
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
<option value="winter">Winter
<option value="spring">Spring
<option value="summer">Summer
<option value="fall">Fall
</select>
```

18.6.2 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
    htp.print('<p>Check Box value: ' || checkboxes(i));
  END LOOP;
END;
/
```

18.6.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 18-10 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 18-10 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) || '.');

    -- If input was OK, stop here. User does not see form again.
    RETURN;
  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>');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  HTP.PRINT('<td valign=center><input type=text name=zip value="' ||
            HTP.ESCAPE_SC(zip) || '">');

  HTP.FORMSUBMIT(NULL, 'Submit');
  HTP.FORMCLOSE;
END;
/
```

18.6.4 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.

18.6.5 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*

18.6.6 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

This technique is best when the user makes a single selection, and the confirmation step of the `Submit` button is not essential.

18.6.7 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.

18.6.8 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> The Oracle Application Server documentation set at:
>
> http://www.oracle.com/technetwork/indexes/documentation/index.html

18.7 Performing Network Operations in PL/SQL Subprograms

Oracle provides packages that allow PL/SQL subprograms to perform a set of network operations using PL/SQL: sending email, getting a host name or address, using TCP/IP connections, retrieving HTTP URL contents, and using table, image maps, cookies, and CGI variables.

Topics:

- Internet Protocol Version 6 (IPv6) Support
- 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

18.7.1 Internet Protocol Version 6 (IPv6) Support

As of Oracle Database 11*g* 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 AI Database PL/SQL Packages and Types Reference* for information about IPv6 support in specific PL/SQL network utility packages

18.7.2 Sending E-Mail from PL/SQL

Using the `UTL_SMTP` package, a PL/SQL subprogram can send e-mail, as in Example 18-11.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `UTL_SMTP` package

Example 18-11 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;
/
```

18.7.3 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

18.7.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `UTL_TCP` package

18.7.5 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 18-12 uses the `UTL_HTTP` package to retrieve the contents of an HTTP URL.

This block shows examples of calls to the procedure in Example 18-12, 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`

For troubleshooting issues related to the `UTL_HTTP` package, see Appendix: Troubleshooting UTL_HTTP.

Example 18-12 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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;

  UTL_HTTP.END_RESPONSE(resp);
    RETURN;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  -- 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;
/
```

18.7.6 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

19

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.

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*.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Call Interface Programmer's Guide* for information about using OCI for CQN

19.1 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 19-1 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 19-1 Query to be Registered for Change Notification

```
SELECT EMPLOYEE_ID, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 10;
```

19.2 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).

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 19-1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Topics:

- Guaranteed Mode
- Best-Effort Mode

> **See Also**
>
> - *Oracle Database Administrator's Guide* for information about the `COMPATIBLE` initialization parameter
> - *Oracle Database Administrator's Guide* for information about AUM

19.2.1 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 19-1 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.

> **See Also**
>
> Queries that Can Be Registered for QRCN in Guaranteed Mode for the characteristics of queries that can be registered in guaranteed mode

19.2.2 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.

The following two examples demonstrate how this works:

- Example: Query Too Complex for QRCN in Guaranteed Mode

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Example: Query Whose Simplified Version Invalidates Objects

19.2.2.1 Example: Query Too Complex for QRCN in Guaranteed Mode

The query in Example 19-2 is too complex for QRCN in guaranteed mode because it contains the aggregate function `SUM`.

Example 19-2 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 this example:

```
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 this example and the transaction in Guaranteed Mode. In best-effort mode, CQN does not simplify the query, but the transaction generates a false positive.

19.2.2.2 Example: Query Whose Simplified Version Invalidates Objects

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 19-3 is too complex for QRCN in guaranteed mode because it includes a subquery.

Example 19-3 Query Whose Simplified Version Invalidates Objects

```
SELECT SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (
  SELECT DEPARTMENT_ID
  FROM DEPARTMENTS
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  WHERE LOCATION_ID = 1700
);
```

In best-effort mode, CQN simplifies the query in this example to this:

```
SELECT * FROM EMPLOYEES, DEPARTMENTS;
```

The simplified query can cause objects to be invalidated. However, if you rewrite the original query as follows, you can register it in either guaranteed mode or best-effort mode:

```
SELECT SALARY
FROM EMPLOYEES, DEPARTMENTS
WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID
  AND DEPARTMENTS.LOCATION_ID = 1700;
```

Queries that can be registered only in best-effort mode are described in Queries that Can Be Registered for QRCN Only in Best-Effort Mode.

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.

19.3 Events that Generate Notifications

These events generate notifications:

- Committed DML Transactions
- Committed DDL Statements
- Deregistration
- Global Events

19.3.1 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.

> **See Also**
>
> ROWID Option

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.3.2 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;
  ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

19.3.3 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.)

A notification is not generated when a client application performs an explicit deregistration.

> **See Also**
>
> Prerequisites for Creating CQN Registrations for privileges required to register queries

19.3.4 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

19.4 Notification Contents

A notification contains some or all of this information:

- Type of event, which is one of:
  - Startup
  - Object change
  - Query result change

Copyright © 1999, 2025, Oracle and/or its affiliates.

  - Deregistration
  - Shutdown
- Registration ID of affected registration
- Names of changed objects
- If `ROWID` option was specified, `ROWID`s of changed rows
- If the notification type is QRCN: Query IDs of queries whose results changed
- If notification resulted from a DML or DDL statement:
  - Array of names of modified tables
  - Operation type (for example, `INSERT` or `UPDATE`)

A notification does not contain the changed data itself. For example, the notification does not say that a monthly salary increased from 5000 to 6000. To obtain more recent values for the changed objects or rows or query results, the application must query the database.

19.5 Good Candidates for CQN

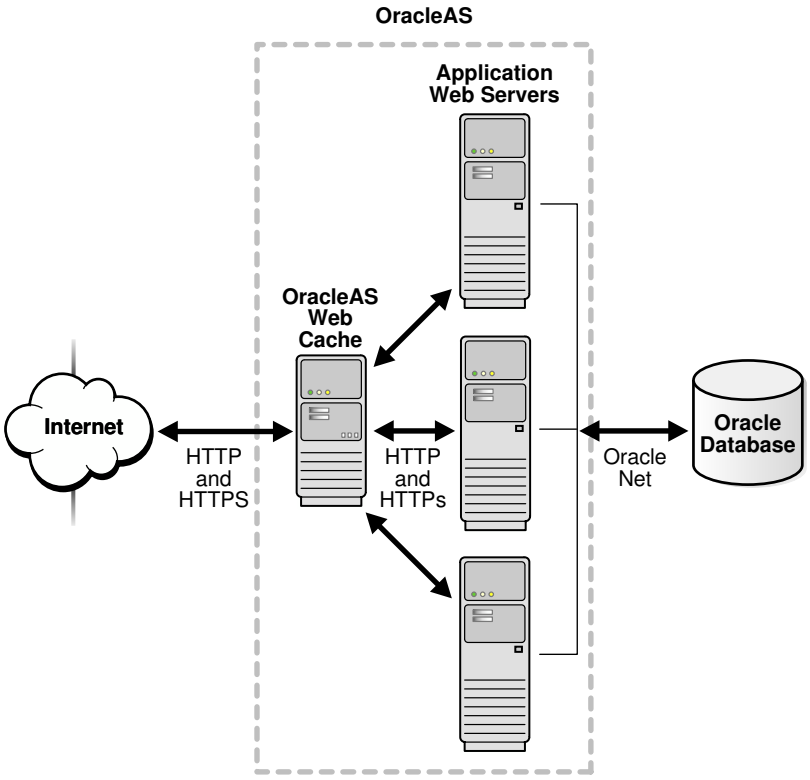
Good candidates for CQN are applications that cache the result sets of queries on infrequently changed objects in the middle tier, to avoid network round trips to the database. These applications can use CQN to register the queries to be cached. When such an application receives a notification, it can 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 it when to refresh the cache.

Figure 19-1 illustrates a typical scenario in which the database serves data that is cached in the middle tier and then accessed over the Internet.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 19-1 Middle-Tier Caching

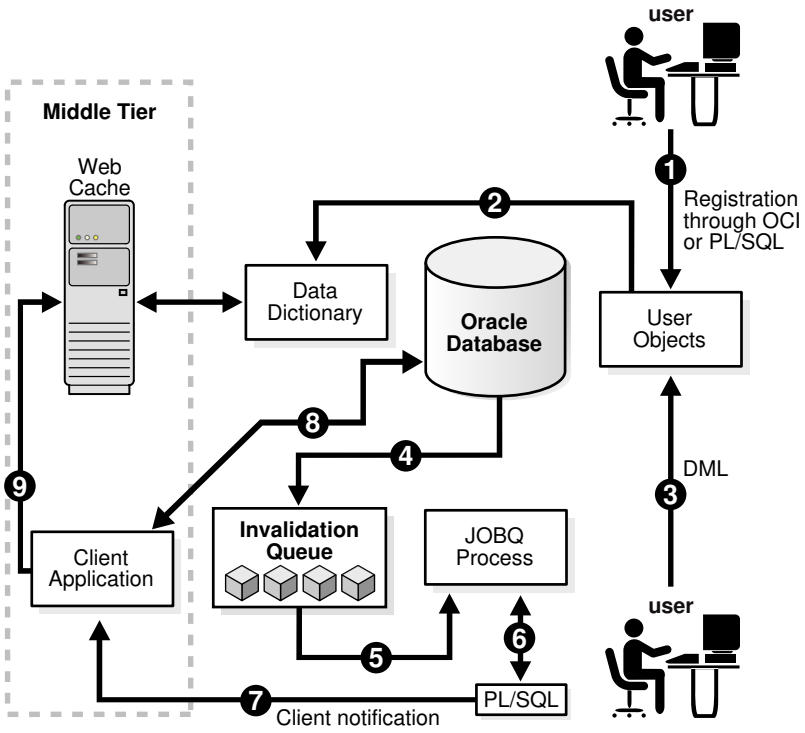


Applications in the middle tier require rapid access to cached copies of database objects while keeping the cache as current as possible in relation to the database. Cached data becomes obsolete when a transaction modifies the data and commits, thereby putting the application at risk of accessing incorrect 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 19-2 illustrates the process by which middle-tier web clients receive and process notifications.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 19-2 Basic Process of Continuous Query Notification (CQN)



Explanation of steps in Figure 19-2 (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 `JOBQ` background process of a notification message.
6. The `JOBQ` process runs the stored procedure specified by the client application. In this example, `JOBQ` passes the data to a server-side PL/SQL procedure. The implementation of the PL/SQL notification handler determines how the notification is handled.
7. Inside the server-side PL/SQL procedure, the developer can implement logic to notify the middle-tier client application of the changes to the registered objects. For example, it notifies the application of the `ROWID` of the changed row in `HR.EMPLOYEES`.
8. The client application in the middle tier queries the back-end database to retrieve the data in the changed row.
9. The client application updates the cache with the data.

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.6 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.

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.
- Oracle Call Interface (OCI)

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

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).

See Also

- About Object Change Notification (OCN)
- About Query Result Change Notification (QRCN)
- Using PL/SQL to Create CQN Registrations
- Using OCI to Create CQN Registrations

19.7 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.7.1 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.

See Also

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_CQ_NOTIFICATION` package
- Using PL/SQL to Register Queries for CQN

19.7.2 CQN Registration Options

You can change the CQN registration defaults with the options summarized in Table 19-1.

Table 19-1 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 `ROWID` 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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Transaction Lag Option (OCN Notification Type Only)
- Notification Grouping Options
- Reliable Option
- Purge-on-Notify and Timeout Options

19.7.2.1 Notification Type Option

The notification types are OCN and QRCN .

> **See Also**
>
> - About Object Change Notification (OCN)
> - About Query Result Change Notification (QRCN)

19.7.2.2 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 Notification Type Option.

QRCN modes are:

- guaranteed
- best-effort

The default is guaranteed mode. For best-effort mode, specify `QOS_BEST_EFFORT` in the `QOSFLAGS` attribute of the `CQ_NOTIFICATION$_REG_INFO` object.

> **See Also**
>
> - Guaranteed Mode
> - Best-Effort Mode

19.7.2.3 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.

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;
```

`ROWID`s 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

If the server does not have enough memory configured for the ROWIDs, the notification might be "rolled up" into a FULL-TABLE-NOTIFICATION, which is 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.
- The UROWID (in the case of an IOT) is larger than 1024 bytes.
- You specified the Notification Grouping option NTFN_GROUPING_TYPE with the value DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY, described in Notification Grouping Options.

Because a FULL-TABLE-NOTIFICATION notification does not include ROWIDs, the application that receives it must assume that the entire table (that is, all rows) might have changed.

You can increase the ROWID threshold using the PL/SQL DBMS_CQ_NOTIFICATION.SET_ROWID_THRESHOLD( ) procedure, which dictates the FULL-TABLE-NOTIFICATION. If the number of rows that are changed exceeds the ROWID threshold, FULL-TABLE-NOTIFICATION is raised.

19.7.2.4 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	DBMS_CQ_NOTIFICATION.INSERTOP
UPDATE	DBMS_CQ_NOTIFICATION.UPDATEOP
DELETE	DBMS_CQ_NOTIFICATION.DELETEOP
ALTEROP	DBMS_CQ_NOTIFICATION.ALTEROP
DROPOP	DBMS_CQ_NOTIFICATION.DROPOP
UNKNOWNOP	DBMS_CQ_NOTIFICATION.UNKNOWNOP
All (default)	DBMS_CQ_NOTIFICATION.ALL_OPERATIONS

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.7.2.5 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 Notification Grouping Options.

The Transaction Lag option specifies the number of transactions by which the client application can lag behind the database. If the number is 0, every transaction that changes a registered object results in a notification. If the number is 5, every fifth transaction that changes a registered object results in a notification. The database tracks intervening changes at object granularity and includes them in the notification, so that the client does not lose them.

A transaction lag greater than 0 is useful only if an application implements flow-of-control notifications. Ensure that the application generates notifications frequently enough to satisfy the lag, so that they are not deferred indefinitely.

If you specify `TRANSACTION_LAG`, then notifications do not include `ROWID`s, even if you also specified `QOS_ROWIDS`.

19.7.2.6 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`	Specifies the type of grouping, which is either of: • `DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY`: All notifications in the group are summarized into a single notification. **Note:** The single notification does not include `ROWID`s, even if you specified the `ROWID` option. • `DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_LAST`: Only the last notification in the group is published and the earlier ones discarded.
`NTFN_GROUPING_START_TIME`	Specifies when to start generating notifications. If specified as `NULL`, it defaults to the current system-generated time.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Attribute	Description
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.

19.7.2.7 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.

19.7.2.8 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.

19.7.3 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.
- You must be connected as a non-SYS user.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

> **See Also**
>
> Deregistration

19.7.4 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
- Queries on tables that are being changed by online redefinition

> **Note**
>
> You can use synonyms in OCN registrations, but not in QRCN registrations.

19.7.5 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.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - Guaranteed Mode and
> - Best-Effort Mode

19.7.5.1 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`).

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;
```

> **Note**
>
> - Sometimes the query optimizer uses an execution plan that makes a query incompatible for guaranteed mode (for example, `OR`-expansion).
> - 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for a list of SQL aggregate functions
> - *Oracle Database SQL Tuning Guide* for information about the query optimizer
> - Best-Effort Mode

19.7.5.2 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`)
  - 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database SQL Language Reference* for a list of SQL functions

19.7.5.3 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 Queries that Can Be Registered for QRCN in Guaranteed Mode
- 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.

> **See Also**
>
> *Oracle Database SQL Tuning Guide* for information about the query optimizer

19.7.6 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.

   See Creating a PL/SQL 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.

   See Creating a CQ_NOTIFICATION$_REG_INFO Object.
3. In your client application, use the `DBMS_CQ_NOTIFICATION.NEW_REG_START` function to open a registration block.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See *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.

4. Run the queries to register. (Do not run DML or DDL operations.)

   See the following topics for more information:

   - Identifying Individual Queries in a Notification
   - Adding Queries to an Existing Registration

5. Close the registration block, using the `DBMS_CQ_NOTIFICATION.REG_END` function.

19.7.6.1 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, *`ntfnds`*, to the notification handler, *`proc_name`*, which handles the notification according to its application requirements. (This is step 6 in Figure 19-2.)

> **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.

> **See Also**
>
> `JOB_QUEUE_PROCESSES`

19.7.6.2 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 19-2 describes the attributes of `SYS.CQ_NOTIFICATION$_REG_INFO`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 19-2 Attributes of CQ_NOTIFICATION$_REG_INFO

Attribute	Description
CALLBACK	Specifies the name of the PL/SQL procedure to be executed when a notification is generated (a notification handler). You must specify the name in the form *schema_name*.*procedure_name*, for example, hr.dcn_callback.
QOSFLAGS	Specifies one or more quality-of-service flags, which are constants in the DBMS_CQ_NOTIFICATION package. For their names and descriptions, see Table 19-3. 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.
OPERATIONS_FILTER	Applies only to OCN (described in About Object Change Notification (OCN)). Has no effect if you specify the QOS_FLAGS attribute with its QOS_QUERY flag. Filters messages based on types of SQL statement. You can specify these constants in the DBMS_CQ_NOTIFICATION package: • ALL_OPERATIONS notifies on all changes • INSERTOP notifies on inserts • UPDATEOP notifies on updates • DELETEOP notifies on deletes • ALTEROP notifies on ALTER TABLE operations • DROPOP notifies on DROP TABLE operations • UNKNOWNOP notifies on unknown operations You can specify a combination of operations with a bitwise OR. For example: DBMS_CQ_NOTIFICATION.INSERTOP + DBMS_CQ_NOTIFICATION.DELETEOP.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 19-2 (Cont.) Attributes of CQ_NOTIFICATION$_REG_INFO

Attribute	Description
TRANSACTION_LAG	**Deprecated**. To implement flow-of-control notifications, use the NTFN_GROUPING_* attributes. Applies only to OCN (described in About Object Change Notification (OCN)). Has no effect if you specify the QOS_FLAGS attribute with its QOS_QUERY flag. Specifies the number of transactions or database changes by which the client can lag behind the database. If 0, then the client receives an invalidation message as soon as it is generated. If 5, then every fifth transaction that changes a registered object results in a notification. 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. 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. If you specify TRANSACTION_LAG, then the ROWID level granularity is unavailable in the notification messages even if you specified QOS_ROWIDS during registration.
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.
NTFN_GROUPING_TYPE	Specifies either of these types of grouping: • DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY: All notifications in the group are summarized into a single notification. • DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_LAST: Only the last notification in the group is published and the earlier ones discarded.
NTFN_GROUPING_START_TIME	Specifies when to start generating notifications. If specified as NULL, it defaults to the current system-generated time.
NTFN_GROUPING_REPEAT_COUNT	Specifies how many times to repeat the notification. Set to DBMS_CQ_NOTIFICATION.NTFN_GROUPING_FOREVER to receive notifications for the life of the registration. To receive at most *n* notifications during the life of the registration, set to *n*.

The quality-of-service flags in Table 19-3 are constants in the DBMS_CQ_NOTIFICATION package. You can specify them with the QOS_FLAGS attribute of CQ_NOTIFICATION$_REG_INFO (see Table 19-2).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 19-3 Quality-of-Service Flags

Flag	Description
QOS_DEREG_NFY	Purges the registration after the first notification.
QOS_RELIABLE	Stores notifications in a persistent database queue. In an Oracle RAC environment, if a database instance fails, surviving database instances can deliver any queued notification messages. **Default:** Notifications are stored in shared memory, which performs more efficiently.
QOS_ROWIDS	Includes the ROWID of each changed row in the notification.
QOS_QUERY	Registers queries for QRCN, described in About Query Result Change Notification (QRCN). If a query cannot be registered for QRCN, an error is generated at registration time, unless you also specify QOS_BEST_EFFORT. **Default:** Queries are registered for OCN, described in About Object Change Notification (OCN)
QOS_BEST_EFFORT	Used with QOS_QUERY. Registers simplified versions of queries that are too complex for query result change evaluation; in other words, registers queries for QRCN in best-effort mode, described in Best-Effort Mode. To see which queries were simplified, query the static data dictionary view DBA_CQ_NOTIFICATION_QUERIES or USER_CQ_NOTIFICATION_QUERIES. These views give the QUERYID and the text of each registered query. **Default:** Queries are registered for QRCN in guaranteed mode, described in Guaranteed Mode

Suppose that you must invoke the procedure HR.dcn_callback whenever a registered object changes. In Example 19-4, 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 19-4 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;
/
```

19.7.6.3 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

`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.

19.7.6.4 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 19-5 adds a query to an existing registration whose registration ID is 21.

Example 19-5 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;
/
```

19.7.7 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.7.8 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``_j`*`number`*`_`*`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 nfcount(cnt NUMBER);
  INSERT INTO nfcount (cnt) VALUES(0);
  COMMIT;
  CREATE OR REPLACE PROCEDURE chnf_callback
  ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  (ntfnds IN CQ_NOTIFICATION$_DESCRIPTOR)
IS
BEGIN
  UPDATE nfcount SET cnt = cnt+1;
  COMMIT;
END;
/
```

- There is a time lag between the commit of a transaction and the notification received by the end user.

19.7.9 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.

19.7.10 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 Creating a PL/SQL Notification Handler.
2. Register the queries on the `HR.EMPLOYEES` and `HR.DEPARTMENTS` tables for QRCN, as instructed in Registering the Queries.

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.

19.7.10.1 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`).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 19-6.

Example 19-6 Creating Server-Side PL/SQL Notification Handler

```
CREATE OR REPLACE PROCEDURE chnf_callback (
  ntfnds IN CQ_NOTIFICATION$_DESCRIPTOR
)
IS
  regid           NUMBER;
  tbname          VARCHAR2(60);
  event_type      NUMBER;
  numtables       NUMBER;
  operation_type  NUMBER;
  numrows         NUMBER;
  row_id          VARCHAR2(2000);
  numqueries      NUMBER;
  qid             NUMBER;
  qop             NUMBER;

BEGIN
  regid := ntfnds.registration_id;
  event_type := ntfnds.event_type;

  INSERT INTO nfevents (regid, event_type)
  VALUES (chnf_callback.regid, chnf_callback.event_type);

  numqueries :=0;

  IF (event_type = DBMS_CQ_NOTIFICATION.EVENT_QUERYCHANGE) THEN
    numqueries := ntfnds.query_desc_array.count;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    FOR i IN 1..numqueries LOOP  -- loop over queries
      qid := ntfnds.query_desc_array(i).queryid;
      qop := ntfnds.query_desc_array(i).queryop;

      INSERT INTO nfqueries (qid, qop)
      VALUES(chnf_callback.qid, chnf_callback.qop);

      numtables := 0;
      numtables := ntfnds.query_desc_array(i).table_desc_array.count;

      FOR j IN 1..numtables LOOP  -- loop over tables
        tbname :=
          ntfnds.query_desc_array(i).table_desc_array(j).table_name;
        operation_type :=
          ntfnds.query_desc_array(i).table_desc_array(j).Opflags;

        INSERT INTO nftablechanges (qid, table_name, table_operation)
        VALUES (
          chnf_callback.qid,
          tbname,
          operation_type
        );

        IF (bitand(operation_type, DBMS_CQ_NOTIFICATION.ALL_ROWS) = 0) THEN
          numrows := ntfnds.query_desc_array(i).table_desc_array(j).numrows;
        ELSE
          numrows :=0;  -- ROWID info not available
        END IF;

        -- Body of loop does not 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;
/
```

19.7.10.2 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 19-7.

Example 19-7 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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
        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 + DBMS_CQ_NOTIFICATION.QOS_ROWIDS,
    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_cq_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 19-6) runs.

Query the table in which notification events are recorded:

```
SELECT * FROM nfevents;
```

Result is similar to:

```
REGID EVENT_TYPE
----- ----------
   61          7
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

```
  QID     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:

```
  QID     TABLE_NAME              ROWID
----- -------------- ------------------
   61 HR.DEPARTMENTS AAANkdAABAAALinAAF
```

19.8 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 for Client Initiated CQN Registrations
- 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

19.8.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

OCI_DTYPE_CHDES

19.8.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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. For Client Initiated CQN using the `OCISubscriptionRegister()` .

> **See Also**
>
> Using OCI for Client Initiated CQN Registrations for more information about Client Initiated CQN.

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.

19.8.3 Using OCI for Client Initiated CQN Registrations

Applications can use client initiated connection mode to register and receive Change Query Notification(CQN).

This mode is designed to work with applications in the cloud but can also be used for applications running on premises. In this mode of notification delivery, the client application initiates a connection to the Oracle database server for receiving notifications. Applications do not require the database server to connect to the application. Client initiated connections do not need special network configuration, are easy to use and secure.

To initiate a CQN registration, your application client can use the OCI interface. Call `OCISubscriptionRegister()` function with the value of the mode set to `OCI_SECURE_NOTIFICATION` to register the application client. You can remove the registration

Copyright © 1999, 2025, Oracle and/or its affiliates.

with `OCISubscriptionUnRegister()` with the value of the mode set to `OCI_SECURE_NOTIFICATION`.

> **See Also**
>
> *Oracle Call Interface Programmer's Guide* for more information about OCISubscriptionRegister() and OCISubscriptionUnRegister()

19.8.4 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:

```
#define OCI_SUBSCR_QOS_RELIABLE            0x01                /* reliable */
#define OCI_SUBSCR_QOS_PURGE_ON_NTFN       0x10      /* purge on first ntfn */
```

- `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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Call Interface Programmer's Guide* for more details about these attributes in publish-subscribe register directly to the database

19.8.5 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 `OCIStmtExecute()`.

> **See Also**
>
> *Oracle Call Interface Programmer's Guide* for more information about `OCI_ATTR_CQ_QUERYID`

19.8.6 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`.

19.8.6.1 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_NFYTYPE` - 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `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`.

19.8.6.1.1 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.

19.8.6.1.2 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Call Interface Programmer's Guide* for more information about continuous query notification descriptor attributes

19.8.7 Demonstrating Continuous Query Notification in an OCI Sample Program

Example 19-8 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 19-8 Program Listing That Demonstrates Continuous Query Notification

```
/* Copyright (c) 2025, 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/<password> 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 AAAMBoAABAAAKX2AAA row_op 4
Query 23 is changed
Table changed is HR.DEPARTMENTS table_op 4
Row changed is AAAMBoAABAAAKX2AAA row_op 4


*The demo program waits for exactly 10 notifications to be received before
*logging off and unregistering the subscription.

*****************************************************************************/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
/*---------------------------------------------------------------------
                    PRIVATE TYPES AND CONSTANTS
  ---------------------------------------------------------------------*/

/*---------------------------------------------------------------------
                    STATIC FUNCTION DECLARATIONS
  ---------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
#include <unistd.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, OCIStmt *stmthp,
                          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, OCIStmt *stmthp,
                              OCIColl *row_changes);
static void processTableChanges(OCIEnv *envhp, OCIError *errhp,
                OCIStmt *stmthp, OCIColl *table_changes);
static void processQueryChanges(OCIEnv *envhp, OCIError *errhp, OCIStmt *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 = NULL, *authp2 = NULL;
  OCIStmt *stmthp, *stmthp2;
  OCIDuration dur, dur2;
  int i;
  dvoid *tmp;
  OCISession *usrhp;
  OCIServer *srvhp;

  printf("Initializing OCI Process\n");
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
/* 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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    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, OCIStmt *stmthp,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                        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, OCIStmt *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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                    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, OCIStmt *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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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;
  OCIStmt *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 **) &srvhp, 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 *) ¬ify_type,
              NULL, OCI_ATTR_CHDES_NFYTYPE, 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( srvhp, 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 *)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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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, usrhp, 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;
OCIStmt *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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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, OCIStmtPrepare (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));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    checker(errhp,
           OCIDefineByPos(stmthp, &defnp3,
                     errhp, 3, (dvoid *)&dname, sizeof(dname),
                    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, OCIStmtExecute (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);

}
```

19.9 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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`

19.10 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

19.10.1 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 19-4 summarizes the attributes of `CQ_NOTIFICATION$_DESCRIPTOR`.

Table 19-4 Attributes of CQ_NOTIFICATION$_DESCRIPTOR

Attribute	Description
`REGISTRATION_ID`	The registration ID that was returned during registration.
`TRANSACTION_ID`	The ID for the transaction that made the change.
`DBNAME`	The name of the database in which the notification was generated.
`EVENT_TYPE`	The database event that triggers a notification. For example, the attribute can contain these constants, which correspond to different database events: • `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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 19-4 (Cont.) Attributes of CQ_NOTIFICATION$_DESCRIPTOR

Attribute	Description
NUMTABLES	The number of tables that were modified.
TABLE_DESC_ARRAY	This field is present only for OCN registrations. For QRCN registrations, it is NULL. If EVENT_TYPE is EVENT_OBJCHANGE]: a VARRAY of table change descriptors of type CQ_NOTIFICATION$_TABLE, each of which corresponds to a changed table. For attributes of CQ_NOTIFICATION$_TABLE, see Table 19-5. Otherwise: NULL.
QUERY_DESC_ARRAY	This field is present only for QRCN registrations. For OCN registrations, it is NULL. If EVENT_TYPE is EVENT_QUERYCHANGE]: a VARRAY of result set change descriptors of type CQ_NOTIFICATION$_QUERY, each of which corresponds to a changed result set. For attributes of CQ_NOTIFICATION$_QUERY, see Table 19-6. Otherwise: NULL.

19.10.2 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 19-5 summarizes the attributes of CQ_NOTIFICATION$_TABLE.

Table 19-5 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: • 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 19-7 describes. If ALL_ROWS was set in the opflags, then the ROW_DESC_ARRAY member is NULL.

Copyright © 1999, 2025, Oracle and/or its affiliates.

19.10.3 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 19-6 summarizes the attributes of CQ_NOTIFICATION$_QUERY.

Table 19-6 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 19-5.

19.10.4 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 19-7 summarizes the attributes of CQ_NOTIFICATION$_ROW.

Table 19-7 Attributes of CQ_NOTIFICATION$_ROW

Attribute	Specifies . . .
OPFLAGS	The type of operation performed on the modified table. See the description of OPFLAGS in Table 19-5.
ROW_ID	The ROWID of the changed row.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

- Choosing a Programming Environment
- Developing Applications with Multiple Programming Languages
- Using Oracle Flashback Technology
- Developing Applications with the Publish-Subscribe Model
- Using the Oracle ODBC Driver
- Using the Identity Code Package
- Microservices Architecture
- Oracle Backend for Microservices and AI
- Developing Applications with Sagas
- Using Lock-Free Reservation
- Developing Applications with Oracle XA
- Developing Applications with Sessionless Transactions
- Understanding Schema Object Dependency
- Using Edition-Based Redefinition
- Using Transaction Guard
- Table DDL Change Notification

> **See Also**
>
> *Oracle AI Database Performance Tuning Guide* and *Oracle AI Database SQL Tuning Guide* for performance issues to consider when developing applications

Copyright © 1999, 2025, Oracle and/or its affiliates.

20

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.

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++).

Topics:

- Overview of Application Architecture
- Overview of the Program Interface
- Overview of PL/SQL
- Overview of Oracle Database Java Support
- Overview of JavaScript
- Choosing PL/SQL or Java or JavaScript
- Overview of Precompilers
- Overview of OCI and OCCI
- Comparison of Precompilers and OCI
- Overview of Oracle Data Provider for .NET (ODP.NET)
- Overview of OraOLEDB

> **See Also**
>
> Developing Applications with Multiple Programming Languages for more information about multilanguage programming

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.1 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

20.1.1 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

20.1.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.1.3 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

20.2 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

20.2.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.2.2 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.

20.3 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

20.4 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 driver that enables data-intensive Java code to run within the database.

Topics:

- Overview of Oracle JVM
- Overview of Oracle JDBC

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Overview of Oracle SQLJ
- Comparison of Oracle JDBC and Oracle SQLJ
- Overview of Java Stored Subprograms
- Overview of Oracle Database Web Services

See Also

- *Oracle AI Database Java Developer's Guide*
- *Oracle AI Database JDBC Developer's Guide*

20.4.1 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 12*c* 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.4.2 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.

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
> - Comparison of Oracle JDBC and Oracle SQLJ

20.4.2.1 Oracle JDBC Drivers

The JDBC standard defines four types of JDBC drivers:

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

20.4.2.1.1 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.

20.4.2.1.2 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 Oracle8*i* or later including Oracle Net, OCI

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

20.4.2.1.3 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.

20.4.2.2 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();
```

20.4.2.3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    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.

20.4.3 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 client-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.

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.

SQLJ is integrated with JDeveloper. Source-level debugging support for SQLJ is available in JDeveloper.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

> **See Also**
>
> *Oracle Database Concepts* for additional general information about SQLJ

20.4.3.1 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, 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.4.4 Comparison of 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.
- PL/SQL and Java stored subprograms can be used interchangeably.

20.4.5 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

20.4.6 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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

20.5 Overview of JavaScript

This section provides you with an overview of how you can run server-side procedural logic in JavaScript using Oracle Multilingual Engine (MLE).

Topics:

- Multilingual Engine Overview
- MLE Concepts
- Understanding MLE Execution Context and Runtime Isolation
- MLE Environment Overview
- JavaScript MLE Modules Overview
- JavaScript MLE Call Specification Overview
- Invoking JavaScript in the Database
- Invoking JavaScript Using MLE Modules
- Invoking JavaScript Using Dynamic MLE Execution
- Privileges for Working with JavaScript in MLE
- Other Supported MLE Features

This section is intended to be an introduction to how you can run JavaScript in Oracle Database. For a more comprehensive guide, see *Oracle Database JavaScript Developer's Guide*.

20.5.1 Multilingual Engine Overview

Oracle Database Multilingual Engine (MLE) enables Oracle Database users to run JavaScript in Oracle Database using stored procedures and dynamic code snippets.

MLE employs a smart database (SmartDB) approach, allowing application logic and data to coexist in the same database. Processing data within the database helps mitigate against problems arising out of handling heavy data volumes and transferring information between servers (database, middle tier, and client).

Using the SmartDB approach, Oracle Database MLE provides these benefits:

- Running the logic in the database removes unnecessary data transfer over the network and improves performance of data-intensive operations.
- Storing and running requirements such as business rules inside the database ensures that every application follows the rules. Hence, implementing security and compliance requirements is simplified.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Storing of commonly used functions in a central place can help with code reuse while avoiding code replication.

If you have data-intensive applications that use up unnecessary resources in a three-tier architecture, you can move the processing logic from the middle tier to the database for faster throughput, better security, and seamless data processing that happens closer to the database.

MLE introduces three kinds of schema objects: MLE modules, MLE environments, and MLE call specifications. MLE provides DDL extensions to create, alter, and drop these objects, and users with the right database privileges can use these extensions. These objects have dictionary views that you can query for information about the objects. MLE also enables you to perform post-execution debugging using the runtime states that are collected during the program runtime.

> **See Also**
>
> Introduction to Multilingual Engine in *Oracle Database JavaScript Developer's Guide* to learn more about JavaScript and MLE.

20.5.2 MLE Concepts

MLE Module

An MLE module is a unit of JavaScript code that is stored in the database as a schema object.

MLE Function

An MLE function is exported by an MLE module and made available for calling from PL/SQL and SQL, as a function or procedure.

MLE Call Specification

An MLE call specification publishes the MLE functions (as PL/SQL functions or procedures) that you can call from SQL and PL/SQL.

MLE Execution Context

An MLE execution context encapsulates all runtime state associated with the execution of JavaScript code.

MLE Environment

An MLE environment configures the properties of MLE execution contexts.

Dynamic MLE

Dynamic MLE uses the `DBMS_MLE` PL/SQL package to enable direct execution of anonymous JavaScript code snippets.

20.5.3 Understanding MLE Execution Context and Runtime Isolation

An MLE execution context is a standalone, isolated runtime environment for JavaScript and encapsulates all runtime state associated with the execution of the JavaScript code in a session.

Copyright © 1999, 2025, Oracle and/or its affiliates.

MLE uses execution contexts to enforce runtime state isolation between MLE modules, anonymous code snippets, and database users. Isolation of different applications in separate contexts within the same session prevents any interference between the applications. Runtime state isolation is important for languages such as JavaScript, where non-local variables have global scope but there is no control over their visibility.

MLE uses execution contexts to ensure that:

- Any code executing in one execution context cannot see or modify runtime state in another execution context.
- All code that shares an execution context has full access to all the runtime states in that context, for example, access to any predefined global variables.
- Different database users cannot observe each other's runtime states.
- All JavaScript code is evaluated using the user, roles, and schema that are in effect at the time of context creation.
- Each context is bound to a single session.

MLE uses execution contexts in two different scenarios:

- With call specifications, where implicit execution contexts are used for SQL and PL/SQL calls to MLE functions
- With dynamic MLE execution, where developers create and use dynamic MLE contexts explicitly

An MLE context has a lifetime that is limited to the session in which it was created. When a session ends or is reset, all the contexts created in that session are dropped. A session can run the JavaScript code as a dynamic MLE snippet or as a function that is exported from an MLE module.

When using multiple contexts, it is recommended that applications create necessary contexts only, and drop contexts when they are not referenced.

> **See Also**
>
> About MLE Execution Contexts in *Oracle Database JavaScript Developer's Guide* for more information about MLE execution contexts.

20.5.4 MLE Environment Overview

You can use an MLE environment to configure specific properties of the execution contexts that MLE uses to run JavaScript code in dynamic MLE snippets and MLE call specifications.

With MLE environments, you can:

- Set JavaScript-specific language options for an execution context at runtime.
- Enable specific MLE modules that you want to import in an execution context. Environments enable developers to create mappings between module imports (`import * from <module-import>`) and modules as stored in the database. For instance, you can create an MLE environment, with "a" mapped to "`module_a`", as in `create mle env myEnv imports('a' module module_a);`.

MLE environments are schema objects that you can manage (create, modify, or drop) and reuse across multiple execution contexts. To create an environment, use the `CREATE MLE ENV`

Copyright © 1999, 2025, Oracle and/or its affiliates.

DDL statement. Alternatively, you can create an environment as an independent copy of an existing environment.

> **Note**
>
> Any change made to the original environment after the cloning, is not propagated to the cloned environment.

Users must have the `CREATE MLE` privilege to create MLE modules and environments in their own schema, or the `CREATE ANY MLE` privilege to create MLE modules and environments in arbitrary schemas.

If no environment is explicitly specified, MLE uses a default environment (from the `SYS` schema) that configures default JavaScript-specific language options and makes built-in MLE modules available for import. The default environment is used for dynamic MLE contexts and module contexts.

Information about MLE environments is available in these dictionary views: `[user | all | dba | cdb]_mle_envs`.

> **See Also**
>
> - Creating MLE Environments in the Database in *JavaScript Developer's Guide* for more information about creating MLE environments
> - CREATE MLE ENV in *Oracle AI Database SQL Language Reference* for more information about the DDL commands used for MLE environments
> - USER_MLE_ENVS, ALL_MLE_ENVS, DBA_MLE_ENVS, and CDB_MLE_ENVS in *Oracle AI Database Reference* for more information about the environment views

20.5.5 JavaScript MLE Modules Overview

You can have JavaScript code stored persistently as an MLE module in the database. An MLE module is a unit of MLE language code that is stored in the database as a schema object and contains code that is written in a single MLE language (in this case, JavaScript). An MLE module enables you to export MLE functions, which you can call from PL/SQL and SQL as a function or procedure.

MLE supports user-defined MLE modules and built-in MLE modules. Built-in modules are not deployed to the database like user-defined MLE modules, but are included as a part of MLE runtime. You cannot change or modify any Oracle-provided JavaScript module.

You can use DDL statements to manage (create, alter, drop) MLE modules as database objects. To get information about MLE modules and other schema objects, use the `USER_MLE_MODULES`, `ALL_MLE_MODULES`, `DBA_MLE_MODULES`, and `CDB_MLE_MODULES` dictionary views.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- Using JavaScript Modules in MLE in *JavaScript Developer's Guide* for more information about MLE modules
- Managing JavaScript MLE Modules
- USER_MLE_MODULES, ALL_MLE_MODULES, DBA_MLE_MODULES, and CDB_MLE_MODULES in *Oracle AI Database Reference* for more information about the module views

20.5.6 JavaScript MLE Call Specification Overview

Call specifications publish JavaScript MLE functions so that they can be called from SQL and PL/SQL. When executing call specifications in a session, MLE loads the module specified in the call specification and calls the function(s) exported by that module.

You can create MLE call specifications using the `CREATE FUNCTION` or `CREATE PROCEDURE` DDL statement syntax with MLE-specific elements.

You can drop a call specification using the `DROP FUNCTION` or `DROP PROCEDURE` DDL statement.

You must have the `CREATE PROCEDURE` privilege to create MLE call specifications in your schema, and the `CREATE ANY PROCEDURE` privilege to create MLE call specifications in an arbitrary schema. You must have the `EXECUTE` privilege on an MLE call specification to call it.

The `MLE MODULE` clause in the `CREATE FUNCTION | PROCEDURE` statement specifies the MLE module that exports the underlying JavaScript function for the call specification. The specified module must always be in the same schema as the call specification being created.

MLE provides three dictionary views that you can query for information about MLE call specifications: `USER_MLE_PROCEDURES`, `ALL_MLE_PROCEDURES`, `DBA_MLE_PROCEDURES`, and `CDB_MLE_PROCEDURES`.

See Also

- Creating an MLE Call Specification in *Oracle Database JavaScript Developer's Guide*
- CREATE PROCEDURE in *Oracle AI Database SQL Language Reference* for more information about the DDL commands used for MLE call specifications
- USER_MLE_PROCEDURES, ALL_MLE_PROCEDURES, DBA_MLE_PROCEDURES, and CDB_MLE_PROCEDURES in *Oracle AI Database Reference* for more information about the environment views

20.5.7 Invoking JavaScript in the Database

You can invoke JavaScript in either of the following ways:

- Using MLE module calls; whereby PL/SQL code references the functions that are exported in JavaScript modules
- Using the `DBMS_MLE` PL/SQL package in Dynamic MLE; whereby you can run JavaScript code snippets using the procedures defined in the package.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- Invoking JavaScript Using MLE Modules
- Invoking JavaScript Using Dynamic MLE Execution

20.5.8 Invoking JavaScript Using MLE Modules

This section provides a brief background of MLE modules and how you can use them to invoke JavaScript in the database.

For the complete documentation related to MLE modules, see *Oracle Database JavaScript Developer's Guide*.

Topics:

- Using MLE Module Contexts
- Specifying an Environment for Call Specifications
- Managing JavaScript MLE Modules
- Running JavaScript Code Using MLE Modules

20.5.8.1 Using MLE Module Contexts

When running a call specification in a session, the JavaScript runtime engine loads the JavaScript module that has the exported functions (JavaScript functions) to be called in the call specification. In any given session, execution contexts for each module called in call specifications are created implicitly on demand. Calls from SQL and PL/SQL to MLE functions are run in a dedicated execution context for the MLE module and for the user on whose behalf the call is executed.

MLE uses execution contexts to maintain runtime state isolation. Call specifications are isolated into separate contexts when they do not share the same user, module, or environment.

Here are some important points to note while using execution contexts with MLE modules:

- Execution contexts separate the runtime state of different users and of different MLE modules, including separate execution contexts in cases where code from a same MLE module is executed on behalf of different database users.
- MLE creates dedicated execution context for each combination of MLE module and MLE environment. Thus, two call specifications that specify either different modules or different environments are executed in separate module contexts, preventing incompatible modules from interfering with each other.
- Within the same session, MLE may employ multiple module contexts to execute call specifications.
- The runtime representation of a module is stateful. A state constitutes elements such as variables in the JavaScript module itself and variables in the global scope that are accessible to the code in the module.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> About MLE Execution Contexts in *JavaScript Developer's Guide* for more information about MLE module execution contexts

20.5.8.2 Specifying an Environment for Call Specifications

An MLE call specification can optionally define the environment for the module context that executes the MLE call specification. To define the environment for a module context, include the `ENV` clause when creating an MLE call specification:

```
CREATE PROCEDURE scott.print_hello(name IN VARCHAR2) AS
 MLE MODULE scott."mymodule"
 ENV scott."myenv"
 SIGNATURE 'printHello';
```

All calls to the scott.print procedure are executed in a module context in which the `scott.mymodule` module is loaded. In addition, the module context is configured according to the `scott.myenv` environment.

Module contexts are separated by environment; if two call specifications refer to the same module but different environments, separate module contexts are created, each configured by their respective environment.

> **See Also**
>
> Specifying Environments for MLE Modules in *JavaScript Developer's Guide* for more information about adding MLE environments for MLE modules

20.5.8.3 Managing JavaScript MLE Modules

You can use SQL DDL statements to create MLE modules as schema objects, provided you have the right privileges. You can use DDL statements to add, alter, or drop JavaScript modules.

Use a `CREATE MLE MODULE` DDL statement to create an MLE module:

Use the `ALTER MLE MODULE` DDL statement to alter attributes of existing modules.

Use the `DROP MLE MODULE` DDL statement to drop modules.

Example 20-1 Creating an MLE Module

```
CREATE MLE MODULE scott."jsmodule"
 LANGUAGE JAVASCRIPT
 AS export function func() { ... }
```

Example 20-2 Replacing an MLE Module

```
CREATE OR REPLACE MLE MODULE scott."jsmodule"
 LANGUAGE JAVASCRIPT
 AS export function func() { ... }
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 20-3 Dropping an MLE Module

```
DROP MLE MODULE scott."jsmodule"
```

> **See Also**
>
> - Using JavaScript Modules in MLE in *JavaScript Developer's Guide* for more information about managing JavaScript MLE modules
> - System and Object Privileges Required for Working with JavaScript in MLE in *JavaScript Developer's Guide* for more information about MLE privileges

20.5.8.3.1 Built-in JavaScript MLE Modules

MLE provides a set of built-in modules that reside in the `SYS` schema. Examples include `mle-js-oracledb` (MLE JavaScript SQL driver), `mle-js-bindings` (used to exchange values between PL/SQL and dynamic MLE contexts), and `mle-js-plsqltypes` (SQL wrapper types definition).

The built-in MLE modules are available for import in any execution context. These modules are not deployed to the database but included as part of the MLE runtime.

> **See Also**
>
> Built-in MLE modules API Documentation

20.5.8.4 Running JavaScript Code Using MLE Modules

You can use JavaScript code in an MLE module in the following ways:

- You can create call specifications to publish the functions that are exported from an MLE module. You can call these MLE functions like you call PL/SQL functions and procedures from SQL and PL/SQL.
- You can use the import statement to import a JavaScript MLE module into other MLE code that is written in JavaScript.

> **See Also**
>
> - Calling MLE JavaScript Functions
> - Importing JavaScript MLE Modules

20.5.8.4.1 Calling MLE JavaScript Functions

From SQL and PL/SQL, you can use MLE call specifications to publish the MLE JavaScript functions. You can call an MLE function from anywhere that you can call a regular PL/SQL function or procedure.

Here is an example:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
create or replace mle module example_module language javascript as
export function string2obj(inputString) {
    if ( inputString === undefined ) {
        throw `must provide a string in the form of key1=value1;...;keyN=valueN`;
    }

    let myObject = {};

    if ( inputString.length === 0 ) {
        return myObject;
    }

    const kvPairs = inputString.split(";");

    kvPairs.forEach( pair => {
        const tuple = pair.split("=");
        if ( tuple.length === 1 ) {
            tuple[1] = false;
        } else if ( tuple.length != 2 ) {
            throw "parse error: you need to use exactly one '=' between key and value
and not use '=' in either key or value";
        }
        myObject[tuple[0]] = tuple[1];
    });
    return myObject;
}
/



create or replace function p_string_to_JSON(p_str varchar2) return JSON
as mle module example_module signature 'string2obj(string)';
/


declare
    l_json   JSON;
    l_string VARCHAR2(100);

begin
    l_string := 'a=1;b=2;c=3;d';
    l_json := p_string_to_JSON(l_string);
    dbms_output.put_line(json_serialize(l_json PRETTY));
end;
/
```

Here is the output:

```
{
  "a" : "1",
  "b" : "2",
  "c" : "3",
  "d" : false
}
```

> **See Also**
>
> Calling MLE JavaScript Functions in *Oracle Database JavaScript Developer's Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.5.8.4.2 Importing JavaScript MLE Modules

You can reuse the functionality in an MLE module in other MLE language code that is outside the MLE module. MLE language code in an existing execution context can import the code of an MLE module (if the code is in the same MLE language) using the language's native import mechanism for example, `import()` in JavaScript.

Module imports in MLE include:

- Importing module functionality in an MLE module into an execution context of another MLE module
- Importing an MLE module into a code snippet to be run in a dynamic MLE execution context

Here is an example of an MLE module importing an MLE function from another MLE module:

```
create or replace mle env default_export_env
    imports ('defaultExportModule' module default_export_module);

create mle module default_import_module language javascript as
import myMathClass from "defaultExportModule";

export function mySum(){
    const result = myMathClass.sum(4, 2);
    console.log(`the sum of 4 and 2 is ${result}`);
}
/
```

> **See Also**
>
> Overview of Importing MLE JavaScript Modules in *JavaScript Developer's Guide*

20.5.9 Invoking JavaScript Using Dynamic MLE Execution

This section provides a brief background of dynamic MLE execution and how you can use it to invoke JavaScript in the database.

For the complete documentation related to dynamic MLE execution, see *Oracle Database JavaScript Developer's Guide*.

Topics:

- Dynamic MLE Execution Overview
- Using Dynamic MLE Execution contexts
- Specifying an Environment for Dynamic MLE Contexts
- Running JavaScript Code Using Dynamic MLE Execution

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.5.9.1 Dynamic MLE Execution Overview

> **Note**
>
> Dynamic MLE execution is suitable for developers working on frameworks (APEX) and server technology (REPL). For all other use cases, using MLE modules and environments to run JavaScript is highly recommended.

As an alternative to using JavaScript MLE modules, the dynamic MLE execution method enables you to run a JavaScript code snippet directly using the `DBMS_MLE` package without having to deploy it as an MLE module first. Dynamic MLE execution is analogous to the `DBMS_SQL` package that is used to execute dynamic SQL.

With dynamic MLE execution, you can invoke a JavaScript code snippet without storing the JavaScript code in the database. The code is not deployed as an MLE module, but is instead provided as `VARACHAR2` or `CLOB` (for larger amounts of code). The code is passed to the `DBMS_MLE` package, which then evaluates and runs the code.

Values can be passed between PL/SQL and dynamic MLE snippets by reading and writing global variables in the appropriate execution context using functions provided in the `DBMS_MLE` package.

> **See Also**
>
> - Running JavaScript Code Using Dynamic MLE Execution
> - Overview of Dynamic MLE Execution in *JavaScript Developer's Guide*

20.5.9.2 Using Dynamic MLE Execution contexts

MLE enables you to have explicit control over which execution context to use for each dynamic MLE snippet. Dynamic MLE execution contexts are created explicitly using the `DBMS_MLE.create_context()` function.

```
FUNCTION create_context(
  environment IN VARCHAR2 DEFAULT NULL
) RETURN context_handle_t;
```

Here are some important points to note while using dynamic MLE execution contexts:

- You can have multiple dynamic MLE snippets evaluated in the same execution context.
- Snippets evaluated in the same context have access to all global variables in the context.
- Snippets evaluated in one context have their execution state completely isolated from snippets evaluated in another context.
- Each dynamic MLE context is created on behalf of a specific database user.
- A dynamic MLE context handle identifies an execution context. This execution context is exclusively used for actions performed on this handle.
- You can create a dynamic execution context using the `DBMS_MLE.create_context()` function, which returns an execution context on behalf of the calling user.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- All MLE code evaluated in a dynamic MLE context executes with the privileges of the user on whose behalf the context was created.
- Each context is bound to a single session. Dynamic MLE snippets in different sessions cannot share runtime state.

> **See Also**
>
> About MLE Execution Contexts in *JavaScript Developer's Guide* for more information about dynamic MLE execution contexts

20.5.9.3 Specifying an Environment for Dynamic MLE Contexts

When creating a dynamic MLE context, you can mention the schema name of an environment object as an optional parameter:

```
DECLARE
  ctx DBMS_MLE.context_handle_t;
BEGIN
  ctx := DBMS_MLE.create_context(environment => 'SCOTT.myenv');
  ...
END;
```

If the environment parameter to the `DBMS_MLE.create_context` function is omitted, the code running in the execution context can import only the MLE built-in modules.

> **Note**
>
> The environment name passed as a string to the `DBMS_MLE.create_context` function is case sensitive and must be a valid schema name. The environment name is not implicitly converted to uppercase.

> **See Also**
>
> *JavaScript Developer's Guide* for more information about environments.

20.5.9.4 Running JavaScript Code Using Dynamic MLE Execution

You can run JavaScript code snippets using the procedures that are defined in the `DBMS_MLE` PL/SQL package.

Using the `DBMS_MLE` package, you can:

- Create dynamic MLE execution contexts.
- Provide JavaScript code snippets as character strings in `VARCHAR2` or `CLOB`.
- Run the code in a dynamic MLE execution context.
- Pass the variables between the PL/SQL and MLE engines; export PL/SQL values to a dynamic MLE execution context, which are then available to code snippets running in that context and import values from a dynamic MLE execution context into PL/SQL.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Deallocate the execution context.

Dynamic MLE snippets can import MLE modules using module-to-module import and execute SQL statements. You must have the `EXECUTE DYNAMIC MLE` privilege to call procedures and functions in the `DBMS_MLE` package. Dynamic MLE execution contexts created using `DBMS_MLE` are created with the privileges of the calling user. Users with the `EXECUTE` privilege on `DBMS_SYS_MLE` also have access to the `DBMS_SYS_MLE` package, which allows these users to create dynamic MLE execution contexts on behalf of any database user.

You can call the `DBMS_MLE.create_context()` function to create an execution context for dynamic MLE code. The function takes an optional parameter that specifies language options, and returns a handle to the created context.

See Also

- Overview of Dynamic MLE Execution in *JavaScript Developer's Guide* for more information about dynamic MLE execution
- Privileges for Working with JavaScript in MLE

20.5.9.4.1 Running JavaScript Code Inline

The following example creates a dynamic MLE execution context and runs a JavaScript snippet in the context using the Q-Quote operator:

```
DECLARE
    l_ctx     dbms_mle.context_handle_t;
    l_snippet CLOB;
BEGIN
    l_ctx := dbms_mle.create_context();
    l_snippet := q'~
// the q-quote operator allows for much more readable code
console.log(`The use of the q-quote operator`);
console.log(`greatly simplifies provision of code inline`);
~';
    dbms_mle.eval(
                 l_ctx,
                 'JAVASCRIPT',
                 l_snippet
    );
    dbms_mle.drop_context(l_ctx);
EXCEPTION
    WHEN OTHERS THEN
        dbms_mle.drop_context(l_ctx);
        RAISE;
END;
/
```

20.5.9.4.2 Running JavaScript Code Using Files

Another method for providing JavaScript code is to read a `CLOB` using a `BFILE` operator, with PL/SQL code segregated from JavaScript code.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Loading JavaScript Code from Files in *JavaScript Developer's Guide* for examples showing how to use files to run JavaScript.

20.5.10 Privileges for Working with JavaScript in MLE

MLE protects access to MLE features using database privileges, which it adds or reuses, as appropriate. Administrators can grant the necessary privileges to users, roles, or both. The minimum privilege that is required to work with MLE JavaScript code is the right to execute JavaScript code in the database. Privileges that allow users to create, alter, or drop MLE schema objects must be restrictive, as must be those with the `ANY` keyword.

To enable you to run any JavaScript code in your own schema, the following object grant must have been issued to your user account:

```
GRANT EXECUTE ON JAVASCRIPT TO <role | user>
```

To include dynamic execution of JavaScript using `DBMS_MLE`, an additional privilege is required:

```
GRANT EXECUTE DYNAMIC MLE TO <role | user>
```

> **Note**
>
> When granting privileges, MLE distinguishes between dynamic MLE execution based on the `DBMS_MLE` package and the MLE execution based on MLE modules and environments.

> **See Also**
>
> - MLE User Privileges for commonly used MLE privileges
> - System and Object Privileges Required for Working with JavaScript in MLE in *JavaScript Developer's Guide* for more information about granting MLE privileges
> - MLE Security in *JavaScript Developer's Guide* for information about other security options available for working with MLE

20.5.10.1 MLE User Privileges

Privilege Name	Description
`CREATE MLE`	Create or replace modules or environments in your schema
`CREATE ANY MLE`	Create or replace modules or environments in any schema
`ALTER ANY MLE`	Alter modules or environments in any schema
`DROP ANY MLE`	Drop modules or environments in any schema
`CREATE PROCEDURE`	Create or replace call specification in your schema
`CREATE ANY PROCEDURE`	Create or replace call specification in any schema

Copyright © 1999, 2025, Oracle and/or its affiliates.

Privilege Name	Description
`DROP ANY PROCEDURE`	Drop call specification in any schema
`EXECUTE DYNAMIC MLE`	Execute any Dynamic MLE functionality
`EXECUTE ON <mle-call-specification>`	Execute the named MLE call specification
`EXECUTE ON <mle-language>`	Execute call specifications or dynamic MLE snippets in the named MLE language
`EXECUTE ON <mle-module>`	Execute call specifications against the referenced MLE module, or define the module as import in an environment
`EXECUTE ON <mle-env>`	Use the named MLE environment to configure a module context or a dynamic MLE context

20.5.11 Other Supported MLE Features

MLE (JavaScript) supports the following other features:

Calling SQL and PL/SQL from the MLE JavaScript SQL Driver

An MLE JavaScript SQL driver enables you to use JavaScript to execute SQL statements and PL/SQL blocks from within JavaScript code. JavaScript has a JavaScript-specific SQL driver API. This API is closely aligned with the API of the corresponding client-side driver. For example, the JavaScript MLE SQL Driver provides an API that is similar to the API of the `node-oracledb` add-on for Node.js.

> **See Also**
>
> Calling PL/SQL and SQL from MLE SQL Driver in *JavaScript Developer's Guide*

Access to `stdout` and `stderr` from JavaScript

MLE provides the functionality to access data written to standard output and error streams from JavaScript code. Within a database session, these streams can be controlled individually for each database user, MLE module, and dynamic MLE context. Information that the executed JavaScript code writes to `stdout` and `stderr` is often valuable for debugging and analysis. The `DBMS_MLE` package allows for mapping `stdout` and `stderr` either to `DBMS_OUTPUT`, or to a user-provided `CLOB`.

> **See Also**
>
> Access to stdout and stderr from JavaScript in *JavaScript Developer's Guide*

SODA Collections in MLE JavaScript Code

You can use any Simple Oracle Document Access (SODA) implementation to perform create, read, update, and delete (CRUD) operations on documents of nearly any kind (including video, image, sound, and other binary content). SODA is a set of NoSQL-style APIs that let you create and store collections of documents (in particular JSON) in Oracle Database, and also let you retrieve them, and query them.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Working with SODA Collections in MLE JavaScript Code in *JavaScript Developer's Guide*

Post-execution Debugging of MLE JavaScript Modules

MLE provides the option to perform post-execution debugging on your JavaScript source code in addition to the standard print debugging. Post-execution debugging allows efficient collection of runtime state during program execution. Once execution of the code has completed, the collected data can be used to analyze program behavior and discover bugs that require attention.

> **See Also**
>
> Post-Execution Debugging of MLE JavaScript Modules in *JavaScript Developer's Guide*

Database Resident Connection Pooling

Database Resident Connection Pooling (DRCP) provides a connection pool in the database server that enables applications to share connections and handle multiple data requests. You can use JavaScript in Oracle Database (MLE) in database connections that use DRCP. In DRCP sessions, you can have applications invoke JavaScript procedures and functions, and use the APIs from the `DBMS_MLE` package. Workloads using in-database JavaScript can leverage the benefits of using DRCP, which include better scalability and improved performance.

> **See Also**
>
> Using Database Resident Connection Pool for more information about using DRCP

20.6 Choosing PL/SQL, Java, or JavaScript

Oracle Database supports PL/SQL, Java, and JavaScript on the server-side and enables you to embed SQL and PL/SQL blocks in your applications using precompilers and APIs on the client-side. To derive maximum benefits from Oracle Database, programmers must choose server-side programming with Oracle Database. With the application logic executing closer to the data, you can avoid network round trips, reuse code with centrally stored functions, and impose business rules within the database to ensure conformity in every application.

Choosing the language to use for your application depends mainly on the type of your application or project, and how you can use a language to your advantage. PL/SQL, Java, and JavaScript have their individual strengths, and they complement each other when used with Oracle Database.

PL/SQL, Java, and JavaScript inter-operate seamlessly in the server through SQL and PL/SQL call signatures. When developing with PL/SQL, you can invoke Java or JavaScript code or reference Java or JavaScript functions from PL/SQL. When developing with Java, you can invoke PL/SQL packages from distributed CORBA or Enterprise Java Beans clients. When

Copyright © 1999, 2025, Oracle and/or its affiliates.

developing with JavaScript, you can call PL/SQL and SQL from within JavaScript using the Multi-lingual Engine (MLE) JavaScript SQL Driver (`mle-js-oracledb`).

PL/SQL packages have their Java and JavaScript equivalents as shown in Table 20-1.

Table 20-1 PL/SQL Packages and Their Java and JavaScript Equivalents

PL/SQL Package	Java Equivalent	JavaScript Equivalent
`DBMS_ALERT`	Call package with JDBC.	Call with `mle-js-oracledb`.
`DBMS_DDL`	JDBC has this functionality.	`mle-js-oracledb` has this functionality.
DBMS_JOB	Schedule a job that has a Java stored subprogram.	Scheduling a job using JavaScript requires a call to `DBMS_SCHEDULER.create_job()` using the built-in JavaScript SQL driver.
`DBMS_LOCK`	Call with JDBC.	Call with `mle-js-oracledb`.
`DBMS_MAIL`	Use JavaMail.	Call with `mle-js-oracledb`.
`DBMS_OUTPUT`	Use subclass `oracle.aurora.rdbms.OracleDBMSOutputStream` or Java stored subprogram `DBMS_JAVA.SET_STREAMS`.	Use `console.log`.
`DBMS_PIPE`	Call with JDBC.	Call with `mle-js-oracledb`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 20-1 (Cont.) PL/SQL Packages and Their Java and JavaScript Equivalents

PL/SQL Package	Java Equivalent	JavaScript Equivalent
DBMS_SESSION	Use JDBC to run an ALTER SESSION statement.	In most cases, you can use mle-js-oracledb to run an ALTER SESSION statement. For example: `myConnection.execute(`alter session set events 'sql_trace wait=true'`);` **Note** Calling ALTER SESSION directly may require additional privileges to change settings, such as session attributes and set events.
DBMS_SNAPSHOT	Call with JDBC.	Call with mle-js-oracledb.
DBMS_SQL	Use JDBC.	Use mle-js-oracledb.
DBMS_TRANSACTION	Use JDBC to run an ALTER SESSION statement.	Use mle-js-oracledb to run an ALTER SESSION statement.
DBMS_UTILITY	Call with JDBC.	Call with mle-js-oracledb.
UTL_FILE	Grant the JAVAUSERPRIV privilege and then use Java I/O entry points.	Call with mle-js-oracledb.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> The `DBMS_JOB` package has been superseded by the `DBMS_SCHEDULER` package, and support for `DBMS_JOB` might be removed in future releases of Oracle Database. In particular, if you are administering jobs to manage system load, you are encouraged to disable `DBMS_JOB` by revoking the package execution privilege for users.
>
> For more information, see DBMS_SCHEDULER and "Moving from `DBMS_JOB` to `DBMS_SCHEDULER`" in the *Oracle Database Administrator's Guide*.

Topics:

- Similarities of PL/SQL, Java, and JavaScript
- Advantages of PL/SQL
- Advantages of Java
- Advantages of JavaScript

20.6.1 Similarities of PL/SQL, Java, and JavaScript

Object-oriented features are common in PL/SQL, Java, and JavaScript as explained here:

- Java is class-oriented and JavaScript has class-based inheritance. PL/SQL does not have an explicit class concept, yet classes can be defined as object types that provide all the features of object-orientation, including constructors, inheritance, polymorphism, and substitution.
- PL/SQL has **type evolution**, which is 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.

Other common aspect of these languages is the packages and libraries that each of them provides to enable you to execute your programs effectively and efficiently.

20.6.2 Advantages of PL/SQL

If you want to build very secure, easy-to-maintain, high performing applications on top of Oracle Database, you must use PL/SQL. As an extension of SQL, PL/SQL supports all SQL data types, data encapsulation, information hiding, overloading, and exception-handling.

The following are the advantages of using PL/SQL:

- SQL data types are easy to use in PL/SQL, with no data type conversions required.
- SQL operations are fast with PL/SQL, especially when a large amount of data is involved, or data validation requirements of your application are high.
- Code development is usually fast in PL/SQL (subject to the development tool or development environment in use). PL/SQL provides package-based stored procedures and functions that perform SQL operations on tables, thereby reducing the network traffic between a program and the database, and saving time for data intensive applications.
- Centralizing application logic provides enhanced security and productivity through built-in APIs, thereby simplifying complex data structures and helping implement security features.
- You can use Pl/SQL to build robust applications using the low-code development platform: Oracle APEX.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- There is a large user base with Oracle-supplied packages and third party libraries that you can draw upon for development.

> **Note**
>
> Some advanced PL/SQL capabilities are unavailable for Java in Oracle9*i* (for example, autonomous transactions and the dblink facility for remote databases).

20.6.3 Advantages of Java

You can use Java if:

- Your application must interact with ERP systems, RMI servers, Java/J2EE, or web services.
- You must develop part of your application in the middle-tier for any the following reasons:
  - Your business logic is complex or compute-intensive with little to moderate direct SQL access.
  - You plan to implement a middle-tier-driven presentation logic.
  - Your application requires transparent Java persistence.
  - Your application requires container-managed infrastructure services.

Thus, when you need 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.

The following are the advantages of using Java:

- You can use Java to create component-based, network-centric application that you can easily update as business needs change.
- You can use Java for open distributed applications, and benefit from many Java-based development tools that are available throughout the industry.
- Java has native mechanisms. For example, Java has built-in security mechanisms, an automatic Garbage Collector, type safety mechanisms, a byte-code verifier, and Java 2 security.
- 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 can use CORBA (which can have many different computer languages in its clients) and Enterprise Java Beans. You can invoke PL/SQL packages from CORBA or Enterprise Java Beans clients.
- You can run XML tools, the Internet File System, or JavaMail from Java.
- You can use Oracle Java Virtual Machine (JVM) for in-place data processing (calling out web services, Hadoop servers, third-party databases, and legacy systems), for running third-party Java libraries, or for running Java-based languages (Jython, Groovy Kotlin, Clojure, Scala, JRuby).
- Java is a robust language in terms of security and can therefore be safely used within the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.6.4 Advantages of JavaScript

If you are looking at ease of scripting to develop end-to-end application with fast execution, JavaScript is your preferred language. With MLE support, you can run stored procedures and user-defined functions written in JavaScript within Oracle Database.

The following are the advantages of using JavaScript:

- With MLE, server-side JavaScript execution is tightly integrated with the database, with code executing within the process of a database session, close to the data and SQL execution. This integration enables efficient data exchange between JavaScript and SQL and PL/SQL.
- When you use JavaScript with Oracle Database, you benefit from the Just-In-Time (JIT) compiler offered by GraalVM, which provides self-optimizing code; so the code adapts to the data that is flowing through it.
- You can use your existing JavaScript code and run them directly against Oracle database without writing the logic in PL/SQL.
- You can create a stored procedure that references a JavaScript module for a function within that module. Alternatively, you can create a stored procedure with the JavaScript code itself inlined within the `CREATE PROCEDURE` or `CREATE FUNCTION` DDL statement.
- JavaScript has the support of a huge community that maintains a vast ecosystem of open-source packages, which you can potentially integrate into your projects.
- It is easy to get started and develop with JavaScript.
- You can use JavaScript as a server-side programming language for your Oracle APEX applications.

20.7 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

20.7.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Pro*C/C++ Developer's Guide* for complete information about the Pro*C/C++ precompiler

Example 20-4 is a code fragment from a C source program that queries the table `employees` in the schema `hr`.

Example 20-4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

20.7.2 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 Developer's Guide* for complete information about the Pro*COBOL precompiler

Example 20-5 is a code fragment from a COBOL source program that queries the table `employees` in the schema `hr`.

Example 20-5 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
     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.

20.8 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> For more information about OCI and OCCI calls:
>
> - *Oracle Call Interface Developer's Guide*
> - *Oracle C++ Call Interface Developer's Guide*
> - *Oracle Database Advanced Queuing User's Guide*
> - *Oracle AI 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

20.8.1 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.

20.8.2 OCI and OCCI Functions

Both OCI and OCCI have four kinds of functions:

Copyright © 1999, 2025, Oracle and/or its affiliates.

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)

20.8.3 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.

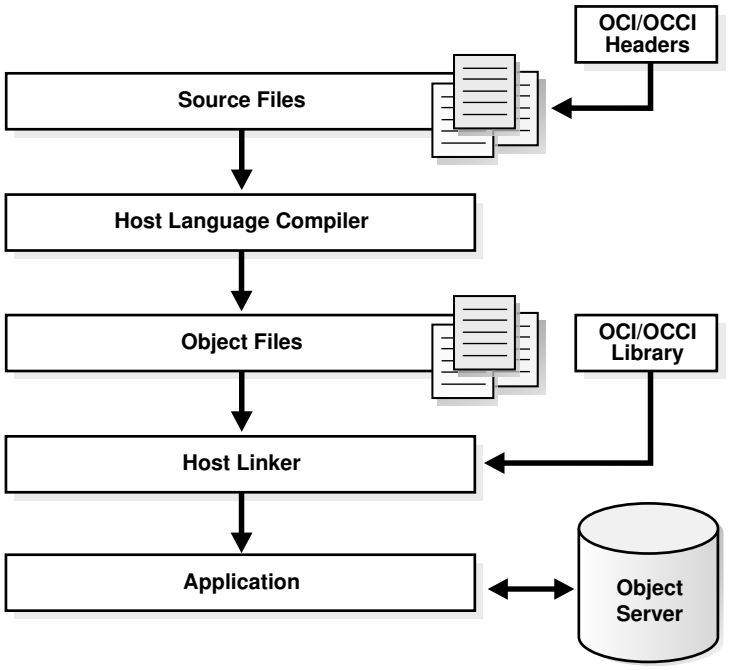
Alternatively, you can use PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. OCI and OCCI also provide facilities for accessing and manipulating objects in Oracle Database.

20.8.4 Building an OCI or OCCI Application

As Figure 20-1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 20-1 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.

20.9 Comparison of Precompilers and 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- OCI calls can be embedded in a Pro*C/C++ application.

20.10 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

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();
  }
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

20.11 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*

Copyright © 1999, 2025, Oracle and/or its affiliates.

21

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

21.1 Overview of Multilanguage Programs

Oracle Database lets you work in different languages:

- PL/SQL, as described in the *Oracle AI Database PL/SQL Language Reference*
- C, through the Oracle Call Interface (OCI), as described in the *Oracle Call Interface Developer's Guide*
- C++, through the Oracle C++ Call Interface (OCCI), as described in the *Oracle C++ Call Interface Developer's Guide*
- C or C++, through the Pro*C/C++ precompiler, as described in the*Pro*C/C++ Developer's Guide*
- COBOL, through the Pro*COBOL precompiler, as described in the*Pro*COBOL Developer'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*

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Java, through the JDBC and SQLJ client-side application programming interfaces (APIs). See *Oracle AI Database JDBC Developer's Guide* and *Oracle AI Database SQLJ Developer's Guide*.
- Java in the database, as described in *Oracle AI 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 AI Database Java Developer's Guide*.

  The Oracle JVM Web Call-Out 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 AI Database Java Developer's Guide*.

How can you choose between these different implementation possibilities? Each of these languages offers different advantages: ease of use, the availability of programmers with specific expertise, the need for portability, and the existence of legacy code are powerful determinants.

The choice might narrow depending on how your application 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.2 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

21.3 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

21.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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
>
> **See Also**
>
> CALLING STANDARD for more information about the calling standard
>
> 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

21.4.1 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)
...
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

21.4.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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.

> **See Also**
>
> *Oracle Call Interface Developer's Guide* for more information about using `agtctl` for `extproc` administration

21.4.3 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`:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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 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/johndoe/dll/myDll.so:/private1/home/johndoe/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.)

21.4.4 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 Publishing External Procedures.

21.5 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

- AS LANGUAGE Clause for Java Class Methods

Copyright © 1999, 2025, Oracle and/or its affiliates.

- AS LANGUAGE Clause for External C Procedures

21.5.1 AS LANGUAGE Clause for Java Class Methods

The `AS LANGUAGE` clause is the interface between PL/SQL and a Java class method.

21.5.2 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.

21.5.2.1 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.

21.5.2.2 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.

21.5.2.3 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.

21.5.2.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.5.2.5 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.

21.5.2.6 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).

21.5.2.7 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.

21.6 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);
   }
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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';
```

21.7 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. */
```

21.8 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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

21.8.1 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;
```

21.8.2 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;
```

21.8.3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      WITH CONTEXT
    --  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE)
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE, SELF)
);
```

21.8.4 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;
```

21.8.5 Example: Java with AUTHID

Here is an example of a publishing a Java class method in a standalone PL/SQL procedure.

```
CREATE OR REPLACE PROCEDURE plsToJ_demoExternal_proc (x PLS_INTEGER, y VARCHAR2, z DATE)
   AUTHID CURRENT_USER
AS LANGUAGE JAVA
   NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
```

21.8.6 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
   NAME "C_demoExternal"
   LIBRARY SomeLib
   WITH CONTEXT
   PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
```

21.8.7 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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   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
      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;
```

21.9 Passing Parameters to External C Procedures with Call Specifications

Call specifications allow a mapping between PL/SQL and C data types.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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

> **See Also**
>
> Specifying Data Types for more information about data type mappings.

21.9.1 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 21-1.

> **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 21-1 Parameter Data Type Mappings

PL/SQL Data Type	Supported External Types	Default External Type

`BINARY_INTEGER`
`BOOLEAN`
`PLS_INTEGER`
	`[UNSIGNED] CHAR`
	`[UNSIGNED] SHORT`
	`[UNSIGNED] INT`
	`[UNSIGNED] LONG`
	`SB1, SB2, SB4`
	`UB1, UB2, UB4`
	`SIZE_T`
		`INT`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 21-1 (Cont.) Parameter Data Type Mappings

PL/SQL Data Type	Supported External Types	Default External Type
NATURAL[1] NATURALN[1] POSITIVE[1] POSITIVEN[1] SIGNTYPE[1]	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	UNSIGNED INT
FLOAT REAL	FLOAT	FLOAT
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR CHARACTER LONG NCHAR NVARCHAR2 ROWID VARCHAR VARCHAR2	STRING OCISTRING	STRING
LONG RAW RAW	RAW OCIRAW	RAW
BFILE BLOB CLOB NCLOB	OCILOBLOCATOR	OCILOBLOCATOR
NUMBER DEC[1] DECIMAL[1] INT[1] INTEGER[1] NUMERIC[1] SMALLINT[1]	OCINUMBER	OCINUMBER
DATE	OCIDATE	OCIDATE
TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime	OCIDateTime
INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	OCIInterval	OCIInterval
composite object types: ADTs	dvoid	dvoid

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 21-1 (Cont.) Parameter Data Type Mappings

PL/SQL Data Type	Supported External Types	Default External Type
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.

21.9.2 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 21-2, 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 21-2 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 *
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 *

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 21-2 (Cont.) 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...
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
SIZE_T	size_t	size_t *	size_t *
SB1	sb1	sb1 *	sb1 *
UB1	ub1	ub1 *	ub1 *
SB2	sb2	sb2 *	sb2 *
UB2	ub2	ub2 *	ub2 *
SB4	sb4	sb4 *	sb4 *
UB4	ub4	ub4 *	ub4 *
FLOAT	float	float *	float *
DOUBLE	double	double *	double *
STRING	char *	char *	char *
RAW	unsigned char *	unsigned char *	unsigned char *
OCILOBLOCATOR	OCILobLocator *	OCILobLocator **	OCILobLocator **
OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
OCIDATE	OCIDate *	OCIDate *	OCIDate *

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 21-2 (Cont.) 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...
OCICOLL	OCIColl * or OCIArray * or OCITable *	OCIColl ** or OCIArray ** or OCITable **	OCIColl ** or OCIArray ** or OCITable **
OCITYPE	OCIType *	OCIType *	OCIType *
TDO	OCIType *	OCIType *	OCIType *
ADT (final types)	dvoid*	dvoid*	dvoid*
ADT (nonfinal types)	dvoid*	dvoid*	dvoid**

Composite 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.

21.9.3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

All IN and RETURN arguments of external types not on this list, all IN OUT arguments, and all OUT arguments are passed by reference.

21.9.4 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.

21.9.5 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.9.6 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 21-3 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 21-3 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 IN OUT OUT RETURN	BY VALUE BY REFERENCE BY REFERENCE BY REFERENCE
LENGTH	[UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG	INT	CHAR LONG RAW RAW VARCHAR2	IN IN OUT OUT RETURN	BY VALUE BY REFERENCE BY REFERENCE BY REFERENCE
MAXLEN	[UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG	INT	CHAR LONG RAW RAW VARCHAR2	IN OUT OUT RETURN	BY REFERENCE BY REFERENCE BY REFERENCE
CHARSETID CHARSETFORM	UNSIGNED SHORT UNSIGNED INT UNSIGNED LONG	UNSIGNED INT	CHAR CLOB VARCHAR2	IN IN OUT OUT RETURN	BY VALUE BY REFERENCE BY REFERENCE BY REFERENCE

In 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      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

21.9.6.1 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 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,

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.9.6.2 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.

21.9.6.3 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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

21.9.6.4 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.

21.9.6.5 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.)

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
{
    OCIString   *NAME;
    OCIDate      B_DATE;
};
typedef struct PERSON PERSON;

struct PERSON_ind
{
    OCIInd    _atomic;
    OCIInd    NAME;
    OCIInd    B_DATE;
};
typedef struct PERSON_ind PERSON_ind;

OCINumber *age (ctx, person_obj, person_obj_ind, tdo, ret_ind)
OCIExtProcContext *ctx;
PERSON         *person_obj;
PERSON_ind     *person_obj_ind;
OCIType        *tdo;
OCIInd         *ret_ind;
{
    sword      err;
    text       errbuf[512];
    OCIEnv    *envh;
    OCISvcCtx *svch;
    OCIError  *errh;
    OCINumber *age;
    int        inum = 0;
    sword      status;

    /* get OCI Environment */
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    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);
}
```

21.9.6.6 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);
```

21.9.6.7 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   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.

21.9.6.8 Interlanguage Parameter Mode Mappings

PL/SQL supports the `IN`, `IN OUT`, and `OUT` parameter modes, and the `RETURN` clause for procedures returning values.

21.10 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.

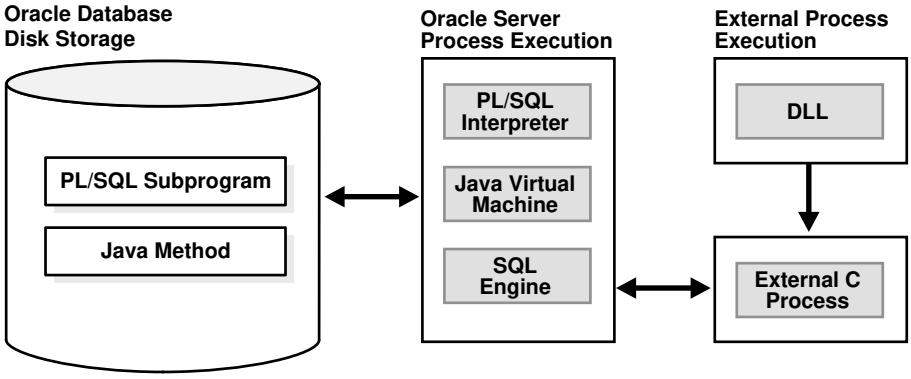
Such calls, which you code in the same manner as a call to a regular PL/SQL procedure, can appear in:

- 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 21-1 shows how Oracle Database and external procedures interact.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 21-1 Oracle Database and External Procedures



Topics:

- Preconditions for External Procedures
- CALL Statement Syntax
- Calling Java Class Methods
- Calling External C Procedures

> **See Also**
>
> CALL Statement Syntax

21.10.1 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

21.10.1.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.10.1.2 Managing Permissions

To call an external procedure, a user must have the EXECUTE privilege on its call specification. To grant this privilege, use the GRANT statement. 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.

> **See Also**
>
> *Oracle Database SQL Language Reference* for more information about GRANT statement

21.10.1.3 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;
```

21.10.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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.

21.10.3 Calling Java Class Methods

To call the `J_calcFactorial` class method published in 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
```

21.10.4 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 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
DECLARE
   g    PLS_INTEGER;
   a    PLS_INTEGER;
   b    PLS_INTEGER;
CALL plsCallsCdivisor_func(a, b);
IF g IN (2,4,8) THEN ...
```

21.11 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.

21.12 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

21.12.1 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

#ifdef LATER
static void checkerr (/*_ OCIError *errhp, sword status _*/);

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                       errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
      *ret_i = (short)OCI_IND_NULL;
      /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
      tmp = OCIExtProcAllocCallMemory(ctx, 1);
      tmp[0] = '\0';
      return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

/*============================================================================*/
int main(char *argv, int argc)
{
  OCIExtProcContext *ctx;
  char           *str1;
  short          str1_i;
  char           *str2;
  short          str2_i;
  short          *ret_i;
  short          *ret_l;
  /* OCI Handles */
  OCIEnv        *envhp;
  OCIServer     *srvhp;
  OCISvcCtx     *svchp;
  OCIError      *errhp;
  OCISession    *authp;
  OCIStmt       *stmthp;
  OCILobLocator *clob, *blob;
  OCILobLocator *Lob_loc;

  /* Initialize and Logon */
  (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                       (dvoid * (*)(dvoid *, size_t)) 0,
                       (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                       (void (*)(dvoid *, dvoid *)) 0 );

  (void) OCIEnvInit( (OCIEnv **) &envhp,
                     OCI_DEFAULT, (size_t) 0,
                     (dvoid **) 0 );

  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                    (size_t) 0, (dvoid **) 0);

  /* Server contexts */
  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                    (size_t) 0, (dvoid **) 0);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  /* 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
```

21.12.2 OCIExtProcRaiseExcp

The `OCIExtProcRaiseExcp` 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
int OCIExtProcRaiseExcp(
   OCIExtProcContext *with_context,
   size_t errnum);
```

The parameters `with_context` and `error_number` are the context pointer and Oracle Database error number. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In SQL*Plus, suppose you publish external procedure `plsTo_divide_proc`, as follows:

```
CREATE OR REPLACE PROCEDURE plsTo_divide_proc (
   dividend IN PLS_INTEGER,
   divisor  IN PLS_INTEGER,
   result   OUT FLOAT)
AS LANGUAGE C
   NAME "C_divide"
   LIBRARY MathLib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      dividend INT,
      divisor  INT,
      result   FLOAT);
```

When called, `C_divide` finds the quotient of two numbers. As this example shows, if the divisor is zero, `C_divide` uses `OCIExtProcRaiseExcp` to raise the predefined exception `ZERO_DIVIDE`:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
{
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise exception ZERO_DIVIDE, which is Oracle Database error 1476. */
    if (OCIExtProcRaiseExcp(ctx, (int)1476) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /* Incorrect parameters were passed. */
      assert(0);
    }
  }
  *result = (float)dividend / (float)divisor;
}
```

21.12.3 OCIExtProcRaiseExcpWithMsg

The `OCIExtProcRaiseExcpWithMsg` service routine raises a user-defined exception and returns a user-defined error message. The C prototype for this function follows:

```
int OCIExtProcRaiseExcpWithMsg(
   OCIExtProcContext *with_context,
   size_t  error_number,
   text   *error_message,
   size_t  len);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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;

}
```

21.13 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

21.13.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The C prototype for this function follows:

```
sword OCIExtProcGetEnv ( OCIExtProcContext *with_context,
   OCIEnv envh,
   OCISvcCtx svch,
   OCIError errh )
```

The parameter `with_context` is the context pointer, and the parameters `envh`, `svch`, and `errh` are the OCI environment, service, and error handles, respectively. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

Both external C procedures and Java class methods can call-back to the database to do SQL operations. For a working example, see Example: Calling an External C Procedure.

> **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);
  ...
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

If you do not use callbacks, you need not include `oci.h`; instead, include `ociextp.h`.

21.13.2 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.

21.13.3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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 `OCIHandleAlloc`, these handle types are not supported:

```
OCI_HTYPE_SERVER
OCI_HTYPE_SESSION
OCI_HTYPE_SVCCTX
OCI_HTYPE_TRANS
```

21.13.4 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 unusually 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

21.13.5 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.

21.13.6 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.

21.13.7 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 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.

> 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

21.13.8 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.
- 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

22

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 Time Travel
- General Guidelines for Oracle Flashback Technology
- Performance Guidelines for Oracle Flashback Technology
- Multitenant Container Database Restrictions for Oracle Flashback Technology

22.1 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 definitional (including schema) changes and 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 the `undo_retention` parameter, 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **Note**
>
> Oracle Database recommends to avoid the usage of `versions_starttime`, `versions_endtime` or `scn_to_timestamp` columns in
>
> `VERSIONS` queries (including `CTAS` queries) to improve the performance.

Topics:

- Application Development Features
- Database Administration Features

> **See Also**
>
> *Oracle Database Concepts* for more information about flashback features

22.1.1 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.

> **See Also**
>
> Using Oracle Flashback Query (SELECT AS OF)

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **See Also**
>
> 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`.

> **See Also**
>
> 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 Also**
>
> 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 Also**
>
> - Flashback Transaction
> - 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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Using DBMS_FLASHBACK Package.

Flashback Time Travel

Use Flashback Time Travel 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.

> **See Also**
>
> Using Flashback Time Travel.

22.1.2 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 Application Development Features.

> **See Also**
>
> - *Oracle Database Administrator's Guide*
> - *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.

22.2 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:

Topics:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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

22.2.1 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`

> **Note**
>
> - Additional configuration of the `UNDO_RETENTION` parameter is required only if you use Oracle Flashback operations or Active Data Guard.
>
> > **See Also**
> >
> > Setting the Minimum Undo Retention Period in *Oracle Database Administrator's Guide* for more information about the `UNDO_RETENTION` parameter
>
> - Starting with Oracle Database release 19c, version 19.9, the value of `UNDO_RETENTION` is not inherited in a CDB.
>
> > **See Also**
> >
> > UNDO_RETENTION in *Oracle Database Reference* for more information about the `UNDO_RETENTION` parameter

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> You can query `V$UNDOSTAT.TUNED_UNDORETENTION` to determine the amount of time for which undo is retained for the current undo tablespace.

Setting `UNDO_RETENTION` does not guarantee that unexpired undo data is not discarded. If the system needs more space, Oracle Database can overwrite unexpired undo with more recently generated undo data.

- Specify the `RETENTION GUARANTEE` clause for the undo tablespace to ensure that unexpired undo data is not discarded.

> **See Also**
>
> - *Oracle Database Administrator's Guide* for more information about creating an undo tablespace and enabling AUM
> - *Oracle Database Reference* for more information about `V$UNDOSTAT`

22.2.2 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;
```

22.2.3 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> If you have very many foreign key constraints, enabling foreign key supplemental logging might not be worth the performance penalty.

22.2.4 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

22.2.5 Granting Necessary Privileges

You or your database administrator must grant privileges to users, roles, or applications that must use these flashback features.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about the `GRANT` statement

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 Time Travel

To allow a specific user to enable Flashback Time Travel on tables, using a specific Flashback Archive, grant the `FLASHBACK ARCHIVE` object privilege on that Flashback Archive to that user.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 Archive, using either the CREATE FLASHBACK ARCHIVE or ALTER FLASHBACK ARCHIVE statement, you must be logged on as SYSDBA.

To disable Flashback Archive for a table that has been enabled for Flashback Archive, you must either be logged on as SYSDBA or have the FLASHBACK ARCHIVE ADMINISTER system privilege.

22.3 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 definitional (DDL) data or schema at a particular time.

  For example, you can issue the flashback query on a table and its schema to view the history of DDL changes made to the table.
- 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 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Topics:

- Example: Examining and Restoring Past Data
- Guidelines for Oracle Flashback Query

> **See Also**
>
> *Oracle Database SQL Language Reference* for more information about the `SELECT AS OF` statement

22.3.1 Example: 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 22-1 retrieves the state of the record for `Chung` at 9:30AM, April 4, 2004:

Example 22-1 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 22-2 restores Chung's information to the `employees` table:

Example 22-2 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'
);
```

22.3.2 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 Time Travel 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.

- 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 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` returns the current date, including fractional seconds.

  `SYSTIMESTAMP` uses the time zone of either the database host system or the database, depending on the setting of the `TIME_AT_DBTIMEZONE` initialization parameter. See Oracle Database Reference: TIME_AT_DBTIMEZONE for more information.
- 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` returns the current date, including fractional seconds.

  `SYSTIMESTAMP` uses the time zone of either the database host system or the database, depending on the setting of the `TIME_AT_DBTIMEZONE` initialization parameter. See Oracle Database Reference: TIME_AT_DBTIMEZONE for more information.
- You can use the `AS OF` clause in queries to check for data that was valid at a particular time.

See Also

- Temporal Validity Support
- Using Flashback Time Travel for information about Flashback Time Travel

22.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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 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 22-1 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 22-1 Oracle Flashback Version Query Row Data Pseudocolumns

Pseudocolumn Name	Description
`VERSIONS_STARTSCN` `VERSIONS_STARTTIME`	Starting System Change Number (SCN) or `TIMESTAMP` when the row version was created. This pseudocolumn identifies the time when the data first had the values reflected in the row version. Use this pseudocolumn to identify the past target time for Oracle Flashback Table or Oracle Flashback Query. If this pseudocolumn is `NULL`, then the row version was created before *start*.
`VERSIONS_ENDSCN`	SCN or `TIMESTAMP` when the row version expired.
`VERSIONS_ENDTIME`	If this pseudocolumn is `NULL`, then either the row version was current at the time of the query or the row corresponds to a `DELETE` operation.
`VERSIONS_XID`	Identifier of the transaction that created the row version.
`VERSIONS_OPERATION`	Operation performed by the transaction: `I` for insertion, `D` for deletion, or `U` for update. The version is that of the row that was inserted, deleted, or updated; that is, the row after an `INSERT` operation, the row before a `DELETE` operation, or the row affected by an `UPDATE` operation. For user updates of an index key, Oracle Flashback Version Query might treat an `UPDATE` operation as two operations, `DELETE` plus `INSERT`, represented as two version rows with a `D` followed by an `I` `VERSIONS_OPERATION`.

A given row version is valid starting at its time `VERSIONS_START*` up to, but not including, its time `VERSIONS_END*`. That is, it is valid for any time *t* such that `VERSIONS_START*` **<= *t* <**

Copyright © 1999, 2025, Oracle and/or its affiliates.

VERSIONS_END*. For example, this output indicates that the salary was 10243 from September 9, 2002, included, to November 25, 2003, excluded.

```
VERSIONS_START_TIME     VERSIONS_END_TIME     SALARY
-------------------     -----------------     ------
09-SEP-2003             25-NOV-2003           10243
```

Here is a typical use of 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.

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
- Using Oracle Flashback Transaction Query

22.5 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 UNDO_SQL code is not the exact opposite of the original transaction. For example, a UNDO_SQL 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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

22.6 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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
VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE empno = 111;
```

Results are similar to:

```
XID               START_SCN    END_SCN O 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 = 'AAATjuAAEAAAAJrAAB';

030002002B210000   10093452   10093459 UPDATE    HR
update "HR"."EMP" set "SALARY" = '555' where ROWID = 'AAATjsAAEAAAAJ7AAA';

030002002B210000   10093452   10093459 BEGIN     HR


4 rows selected.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

22.7 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

22.7.1 Using Flashback Version Query with DBMS_FLASHBACK

You can enable DBMS_FLASHBACK for a session using the DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER or DBMS_FLASHBACK.ENABLE_AT_TIME procedure. The ENABLE_AT_SYSTEM_CHANGE_NUMBER and ENABLE_AT_TIME procedures are based on System Change Number (SCN). You can use Oracle Flashback Version Query (version query) to retrieve committed row versions pertaining to a time interval. To use a version query (with the VERSIONS BETWEEN clause), you need to specify the lower and upper SCN limits, either using specific SCN values or using the MINVALUE and MAXVALUE keywords.

Using a version query along with DBMS_FLASHBACK affects the MAXVALUE keyword or the upper SCN limit in the version query, in that if the version query's upper SCN limit (through the MAXVALUE keyword or provided value) exceeds the DBMS_FLASHBACK SCN, the version query takes the DBMS_FLASHBACK SCN as the maximum limit. For example, suppose that the session-level Flashback is enabled using the ENABLE_AT_SYSTEM_CHANGE_NUMBER procedure with the SCN parameter set as SCN_X. A version query using VERSIONS BETWEEN clause is passed later with the minimum and maximum limits as SCN_A and SCN_B. When the version query runs, the SCNs are treated in the following manner:

- When the versions query specifies a MAXVALUE clause (instead of a specific SCN_B), SCN_X is treated as the maximum limit.
- When the versions query has two SCNs, SCN_A and SCN_B, and SCN_B is less than SCN_X, then the SCNs in the versions query are retained without any changes.
- When the versions query has two SCNs, SCN_C and SCN_D,and SCN_D is greater than SCN_X, then SCN_X is treated as the maximum limit.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The following code sample shows how a version query could be used after DBMS_FLASHBACK is enabled for a session.

```
DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER(123);
…
…
SELECT * FROM <table_name> VERSIONS BETWEEN SCN 120 and 125;
…
DBMS_FLASHBACK.DISABLE();
```

Here, the upper SCN limit of 125 in the version query is replaced with DBMS_FLASHABCK SCN of 123 because the version query's upper SCN limit is greater than the DBMS_FLASHABCK SCN. Therefore, the following is the resultant version query:

```
SELECT * FROM <table_name> VERSIONS BETWEEN SCN 120 and 123;
```

See Also

- *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_FLASHBACK package
- *Oracle Database SQL Language Reference* for information about Oracle Flashback Version Query

22.8 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

22.8.1 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.

22.8.2 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 22-2

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 22-2 Flashback TRANSACTION_BACKOUT Options

Option	Description
CASCADE	Backs out specified transactions and all dependent transactions in a post-order fashion (that is, children are backed out before parents are backed out). Without CASCADE, if any dependent transaction is not specified, an error occurs.
NOCASCADE	Default. Backs out specified transactions, which are expected to have no dependent transactions. First dependent transactions causes an error and appears in *_FLASHBACK_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.

See Also

Oracle Database PL/SQL Packages and Types Reference for syntax of the TRANSACTION_BACKOUT procedure and detailed parameter descriptions

22.8.3 TRANSACTION_BACKOUT Reports

To see the reports that TRANSACTION_BACKOUT generates, query the static data dictionary views *_FLASHBACK_TXN_STATE and *_FLASHBACK_TXN_REPORT.

22.8.3.1 *_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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database Reference* for more information about *_FLASHBACK_TXN_STATE

22.8.3.2 *_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

22.9 Using Flashback Time Travel

Flashback Time Travel provides the ability to track and store definitional (including schema) and transactional changes to a table over its lifetime.

Using Flashback Time Travel, you can enable tracking of DML (such as INSERT and DELETE) and DDL operations (such as creating and truncating tables) on a table that is being tracked (also called tracked table). You can then use Flashback Data Archive (also called Flashback Archive) to archive the changes made to the rows of the tracked table in history tables. Flashback Time Travel also maintains a history of the evolution of a table schema. Having the history of the table and schema enables you to issue flashback queries (AS OF and VERSIONS) on the table and its schema. You can also view the history of DDL and DML changes made to the table.

With the Flashback Time Travel feature, you can create several Flashback Archives in your database. A Flashback Archive is a logical entity that is associated with a set of tablespaces. There is a quota that is reserved for the archives on those tablespaces and a retention period for the archived data. Using a Flashback Archive improves performance and helps in complying with record stage policies and audit reports.

A Flashback Archive consists of one or more tablespaces or parts thereof. You can have multiple Flashback Archives. If you are logged on as SYSDBA, you can specify a default Flashback Archive for the system. A Flashback Archive is configured with retention time. Data archived in the Flashback Archive is retained for the retention time that is specified when creating the Flashback Archive.

When choosing a Flashback Archive for a specific table, consider the data retention requirements for the table and the retention times of the Flashback Archives on which you have the FLASHBACK ARCHIVE object privilege.

To ensure data security of the Flashback Archive history tables, when enabling a Flashback Archive for a table, you can specify that a tracked table should use a blockchain history table for the Flashback Archive.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Protecting Flashback Archive Data for more information about using blockchain history tables for the Flashback Archive

By default, Flashback Archive is not enabled for any table. Consider enabling Flashback Archive for user context tracking and database hardening.

- **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.

- **Database hardening**. You can associate a set of tables together in an "application", and then enable Flashback Archive on all those tables with a single command. Database hardening also 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 Time Travel 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*.

You can also use Flashback Time Travel in various scenarios, such as enforcing digital shredding, accessing historical data, selective data recovery, and auditing.

Flashback Time Travel Restrictions

- You cannot enable Flashback Archive on tables with `LONG` data type or nested table columns.
- You cannot enable Flashback Archive on a nested table, temporary table, external table, materialized view, Advanced Query (AQ) table, hybrid partitioned tables, or non-table object.
- Flashback Archive does not support DDL statements that move, split, merge, or coalesce partitions or sub partitions, move tables, or convert `LONG` columns to LOB columns.
- Adding or enabling a Constraint (including Foreign Key Constraint) on a table that has been enabled for Flashback Archive fails with ORA-55610. Dropping or disabling a Constraint (including Foreign Key Constraint) on a table that has been enabled for Flashback Archive is supported.
- After enabling Flashback Archive on a table, Oracle recommends initially waiting at least 20 seconds before inserting data into the table and waiting up to 5 minutes before using Flashback Query on the table.
- Dropping a Flashback Archive base table requires Flashback Archive on the base table to be disabled first, and then the base table can be dropped. Disabling Flashback Archive will remove the historical data, while disassociating the Flashback Archive will retain the historical data. Truncate of the base table, on the other hand, is supported and the historical data will remain available in the Flashback Archive.
- If you enable Flashback Archive on a table, but Automatic Undo Management (AUM) is disabled, error ORA-55614 occurs when you try to modify the table.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- You cannot enable Flashback Archive if the table use any of these Flashback Time Travel reserved words as column names: `STARTSCN`, `ENDSCN`, `RID`, `XID`, `OP`, `OPERATION`.
- You can expect the Flashback Archive operations to slow down or become unresponsive when its space usage exceeds 90% of the maximum space for each of the available tablespaces that it is associated with. Ensure that you increase the maximum space that the Flashback Archive can use if the Flashback Archive usage nears 90% threshold for all associated tablespaces.
- In the standard mode (`MAX_COLUMNS = STANDARD`), a table that is enabled for Flashback Archive allows a maximum of 991 columns as apposed to a maximum of 1024 columns that are allowed for non-archived tables.
- In the extended mode (`MAX_COLUMNS = EXTENDED`), a table that is enabled for Flashback Archive allows a maximum of 4087 columns as opposed to a maximum of 4096 columns that are allowed for non-archived tables.

Topics:

- DDL Statements on Tables Enabled for Flashback Archive
- Creating a Flashback Archive
- Altering a Flashback Archive
- Dropping a Flashback Archive
- Specifying the Default Flashback Archive
- Enabling and Disabling Flashback Archive
- Viewing Flashback Archive Data
- Transporting Flashback Archive Data between Databases
- Flashback Time Travel Scenarios
- Protecting Flashback Archive Data

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_FLASHBACK_ARCHIVE` package

22.9.1 DDL Statements on Tables Enabled for Flashback Archive

Flashback 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 Archive does not support DDL statements that move, split, merge, or coalesce partitions or subpartitions, move tables, or convert `LONG` columns to `LOB` columns.

Copyright © 1999, 2025, Oracle and/or its affiliates.

For example, the following DDL statements cause error ORA-55610 when used on a table enabled for Flashback 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 Archive, use the `DBMS_FLASHBACK_ARCHIVE.DISASSOCIATE_FBA` procedure to disassociate the base table from its Flashback Archive. To reassociate the Flashback Archive with the base table afterward, use the `DBMS_FLASHBACK_ARCHIVE.REASSOCIATE_FBA` procedure. Also, to drop a table enabled for Flashback Archive, you must first disable Flashback 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

22.9.2 Creating a Flashback Archive

You can create a Flashback Archive with the `CREATE FLASHBACK ARCHIVE` statement.

Create a Flashback Archive with the `CREATE FLASHBACK ARCHIVE` statement, specifying:

- Name of the Flashback Archive
- Name of the first tablespace of the Flashback Archive
- (Optional) Maximum amount of space that the Flashback 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.

Note

Ensure that you provide enough space as the maximum space for the Flashback Archive. If the used space exceeds 90% of the maximum space for each of the available Flashback Archive tablespaces, the Flashback Archive operations can slow down or become unresponsive.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Retention time (number of days that Flashback 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 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 Archive for the system. If you omit this option, you can still make this Flashback Archive as default at a later stage.

Oracle recommends that all users who must use Flashback Archive have unlimited quota on the Flashback 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 Archive named `fla1` that uses up to 10 G of tablespace `tbs1`, whose data are retained for one year:

```
CREATE FLASHBACK ARCHIVE DEFAULT fla1 TABLESPACE tbs1
  QUOTA 10G RETENTION 1 YEAR;
```

- Create a Flashback Archive named `fla2` that uses tablespace `tbs2`, whose data are retained for two years:

```
CREATE FLASHBACK ARCHIVE fla2 TABLESPACE tbs2 RETENTION 2 YEAR;
```

> **See Also**
>
> - *Oracle Database SQL Language Reference* for more information about the `CREATE FLASHBACK ARCHIVE` statement syntax
> - Specifying the Default Flashback Archive

22.9.3 Altering a Flashback Archive

You can modify a Flashback Archive using the `ALTER FLASHBACK ARCHIVE` statement.

With the `ALTER FLASHBACK ARCHIVE` statement, you can:

- Change the retention time of a Flashback Archive
- Purge some or all of its data
- Add, modify, and remove tablespaces

> **Note**
>
> Removing all tablespaces of a Flashback 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 Archive for the system.

Examples

- Make Flashback Archive `fla1` the default Flashback Archive:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
ALTER FLASHBACK ARCHIVE fla1 SET DEFAULT;
```

- To Flashback Archive `fla1`, add up to 5 G of tablespace `tbs3`:

  ```
  ALTER FLASHBACK ARCHIVE fla1 ADD TABLESPACE tbs3 QUOTA 5G;
  ```

- To Flashback Archive `fla1`, add as much of tablespace `tbs4` as needed:

  ```
  ALTER FLASHBACK ARCHIVE fla1 ADD TABLESPACE tbs4;
  ```

- Change the maximum space that Flashback Archive `fla1` can use in tablespace `tbs3` to 20 G:

  ```
  ALTER FLASHBACK ARCHIVE fla1 MODIFY TABLESPACE tbs3 QUOTA 20G;
  ```

- Allow Flashback Archive `fla1` to use as much of tablespace `tbs1` as needed:

  ```
  ALTER FLASHBACK ARCHIVE fla1 MODIFY TABLESPACE tbs1;
  ```

- Change the retention time for Flashback Archive `fla1` to two years:

  ```
  ALTER FLASHBACK ARCHIVE fla1 MODIFY RETENTION 2 YEAR;
  ```

- Remove tablespace `tbs2` from Flashback Archive `fla1`:

  ```
  ALTER FLASHBACK ARCHIVE fla1 REMOVE TABLESPACE tbs2;
  ```

  (Tablespace `tbs2` is not dropped.)

- Purge all historical data from Flashback Archive `fla1`:

  ```
  ALTER FLASHBACK ARCHIVE fla1 PURGE ALL;
  ```

- Purge all historical data older than one day from Flashback Archive `fla1`:

  ```
  ALTER FLASHBACK ARCHIVE fla1
    PURGE BEFORE TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY);
  ```

- Purge all historical data older than SCN 728969 from Flashback Archive `fla1`:

  ```
  ALTER FLASHBACK ARCHIVE fla1 PURGE BEFORE SCN 728969;
  ```

> **See Also**
>
> *Oracle Database SQL Language Reference* for more information about the `ALTER FLASHBACK ARCHIVE` statement

22.9.4 Dropping a Flashback Archive

You can drop a Flashback Archive with the `DROP FLASHBACK ARCHIVE` statement.

Dropping a Flashback Archive deletes its historical data, but does not drop its tablespaces.

Example

Remove Flashback Archive `fla1` and all its historical data, but not its tablespaces:

```
DROP FLASHBACK ARCHIVE fla1;
```

Oracle Database SQL Language Reference for more information about the `DROP FLASHBACK ARCHIVE` statement syntax

Copyright © 1999, 2025, Oracle and/or its affiliates.

22.9.5 Specifying the Default Flashback Archive

You can specify the default Flashback Archive using the CREATE or ALTER FLASHBACK ARCHIVE statements.

The default Flashback Archive for the system is the default Flashback Archive for every user who does not have their own default Flashback Archive.

By default, the system has no default Flashback Archive. If you are logged on as SYSDBA, you can specify default Flashback Archive in either of these ways:

- Specify the name of an existing Flashback 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 Archive. For example:

  ```
  CREATE FLASHBACK ARCHIVE DEFAULT fla2 TABLESPACE tbs1
    QUOTA 10G RETENTION 1 YEAR;
  ```

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

22.9.6 Enabling and Disabling Flashback Archive

By default, Flashback Archive is disabled for all table. You can enable Flashback Archive for a table if you have the FLASHBACK ARCHIVE object privilege on the Flashback Archive to use for that table.

To enable Flashback 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 Archive where the historical data for the table are stored. The default is the default Flashback Archive for the system. If you specify a nonexistent Flashback Archive, an error occurs.

If a table has Flashback Archive enabled, and you try to enable it again with a different Flashback Archive, an error occurs.

After Flashback 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 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.)

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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 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 Archive `fla1`:

```
CREATE TABLE employee (EMPNO NUMBER(4) NOT NULL, ENAME VARCHAR2(10),
  JOB VARCHAR2(9), MGR NUMBER(4)) FLASHBACK ARCHIVE fla1;
```

- Enable Flashback Archive for the table `employee` and store the historical data in the default Flashback Archive:

```
ALTER TABLE employee FLASHBACK ARCHIVE;
```

- Enable Flashback Archive for the table `employee` and store the historical data in the Flashback Archive `fla1`:

```
ALTER TABLE employee FLASHBACK ARCHIVE fla1;
```

- Disable Flashback Archive for the table `employee`:

```
ALTER TABLE employee NO FLASHBACK ARCHIVE;
```

22.9.7 Viewing Flashback Archive Data

You can view information about Flashback Archive files in static data dictionary views.

Table 22-3 Static Data Dictionary Views for Flashback Archive Files

View	Description
`*_FLASHBACK_ARCHIVE`	Displays information about Flashback Archive files
`*_FLASHBACK_ARCHIVE_TS`	Displays tablespaces of Flashback Archive files
`*_FLASHBACK_ARCHIVE_TABLES`	Displays information about tables that are enabled for Flashback Archive

> **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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

22.9.8 Transporting Flashback Archive Data between Databases

You can export and import the Flashback Archive base tables along with their history to another database using the `FLASHBACK_ARCHIVE_MIGRATE` package and the Oracle Transportable Tablespaces capability.

DBMS_FLASHBACK_ARCHIVE_MIGRATE enables the migration of Flashback Archive enabled tables from a database on any release in which the package exists to any database on any release that supports Flashback Time Travel.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about using the DBMS_FLASHBACK_ARCHIVE_MIGRATE package

22.9.9 Flashback Time Travel Scenarios

- Scenario: Using Flashback Time Travel to Enforce Digital Shredding
- Scenario: Using Flashback Time Travel to Access Historical Data
- Scenario: Using Flashback Time Travel to Generate Reports
- Scenario: Using Flashback Time Travel for Auditing
- Scenario: Using Flashback Time Travel to Recover Data

22.9.9.1 Scenario: Using Flashback Time Travel 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 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.)

22.9.9.2 Scenario: Using Flashback Time Travel 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 Archive named `fla1` 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 fla1 TABLESPACE tbs1
  QUOTA 10G RETENTION 5 YEAR;
```

Enable Flashback Archive for the tables `inventory` and `stock_data`, and store the historical data in the default Flashback Archive:

```
ALTER TABLE inventory FLASHBACK ARCHIVE;
ALTER TABLE stock_data FLASHBACK ARCHIVE;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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;
```

22.9.9.3 Scenario: Using Flashback Time Travel 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 Archive named `fla2` 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 fla2 TABLESPACE tbs1
  QUOTA 20G RETENTION 5 YEAR;
```

Enable Flashback Archive for the table `investments`, and store the historical data in the default Flashback 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';
```

22.9.9.4 Scenario: Using Flashback Time Travel 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 Archive named `fla4` that uses up to 100 G of tablespace `tbs1`, whose data are retained for 10 years:

```
CREATE FLASHBACK ARCHIVE DEFAULT fla4 TABLESPACE tbs1
  QUOTA 100G RETENTION 10 YEAR;
```

The company enables Flashback Archive for the table `Billings`, and stores the historical data in the default Flashback 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');
```

22.9.9.5 Scenario: Using Flashback Time Travel 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

required historical information is available in the Flashback 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 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 runs again the update that Lisa had requested.

22.9.10 Protecting Flashback Archive Data

You can choose to protect the Flashback Archive (FBA) data for a tracked table (a user table whose changes need tracking). To do so, when creating a table (non-blockchain, non-immutable table), use the optional `BLOCKCHAIN` keyword with the `FLASHBACK ARCHIVE` clause. The archived data is then stored in an Oracle-managed blockchain log history table.

The following example code uses the `BLOCKCHAIN FLASHBACK ARCHIVE` clause with the `CREATE TABLE` command to create a table that is tracked using blockchain-based FBA.

```
CREATE TABLE tab1 (
  id           NUMBER,
  description  VARCHAR2(50),
  CONSTRAINT tab_1_pk PRIMARY KEY (id)
) BLOCKCHAIN FLASHBACK ARCHIVE fba_1year;
```

> **Note**
>
> Ensure that the FBA that you specify already exists before creating the table.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> CREATE TABLE *flashback_archive_clause* and CREATE FLASHBACK ARCHIVE in *Oracle Database SQL Language Reference* for more information about designating a table as a tracked table for FBA

Having an FBA history table as a blockchain table enables you to record the history of changes made to regular user tables (that are tracked) in a cryptographically secure and verifiable manner. You can query the FBA history tables to determine if there is any tampering of the tracked table's content.

Blockchain Log History Table Overview

Blockchain Log history tables, like any other blockchain tables, are append-only tables that organize rows into a number of chains, with each row in a chain (except the first row) chained to the previous row in the chain using a cryptographic hash. Each change in a tracked table is added to the blockchain table as a separate row within the cryptographic hash chain that is created and maintained in the blockchain table.

> **See Also**
>
> Managing BlockchainTables in *Oracle Database Administrator's Guide* for more information about blockchain tables

The following points summarize a blockchain log history table (blockchain history table).

- A blockchain history table is an Oracle-managed, append-only table with an Oracle-defined schema definition.
- A blockchain history table is used to track the changes in a user table and to ensure that the changes are tamper evident.
- You can verify the data and chain integrity of a blockchain history table using the standard blockchain table APIs.
- Blockchain tables have associated table-level and row-level retention periods. To determine the row retention period to use for the blockchain history tables, the higher of the row retention periods of the FBA history table and the blockchain table is used. Similarly, to determine the table retention period, the higher of the table retention periods of the FBA history table and the blockchain table is used.
- Row deletion in a blockchain history table is only allowed for a prefix of the chains.
- The owner of a blockchain history table is the same as that of the FBA history table.

Understanding How FBA Data is Stored

A row of a tracked table transitions through a series of lifespans, with each lifespan created whenever transactions commit changes to the row. The most recent lifespan (created by the last transaction that modified the row) is referred to as the "current lifespan," and all the previous lifespans are referred to as "historical lifespans."

Historical lifespans have a start System Change Number (SCN) and an end SCN, denoting the start and end time for the lifespan of a particular row version in a table. Current lifespans have

Copyright © 1999, 2025, Oracle and/or its affiliates.

a start SCN, but the end SCN is infinite. The end SCN of the newly created historical lifespan is set to the commit SCN of the latest transaction that affected the row.

Flashback Time Travel uses the following internal tables to record the lifespans from each tracked table:

- A blockchain history table (`SYS_FBA_HIST_<objno>`) to record the historical lifespans
- A temporary table (`SYS_FBA_TCRV_<objno>` or `tcrv` table) to record the current lifespan

When you enable FBA for a table, Flashback Time Travel internally creates the corresponding history and temporary (`tcrv`) tables. The blockchain history table stores all the historical row versions whenever a change happens on a row of the tracked table. The `tcrv` table (not a blockchain table) records the most recent row changes.

> **See Also**
>
> Appendix: Recording DML Changes on the Tracked Table in *Appendices* for more information about how the DML updates to the tracked table are recorded in blockchain and temporary tables

Protecting FBA Data and Verifying its Integrity

Using blockchain tables for FBA history tables enables you to replay the secure history from the FBA to determine whether there has been any tampering in the tracked tables.

Blockchain works by computing and saving a cryptographic digest (blockchain digest) of the data. A blockchain digest is a well-defined summary of a blockchain table that is digitally signed by the Oracle instance using the table owner's private key. Computing a blockchain digest provides a snapshot of the metadata and data about the last row in each chain of a blockchain table at any particular time. This digest can be saved outside the database to detect tampering. If the data you are tracking has changed, then the blockchain digest of the new data differs from the previously stored digest.

> **See Also**
>
> Format of Signed Digest in *Oracle Database Administrator's Guide* that describes the data format for the blockchain table digest

Blockchain tables are highly secure because they automatically chain new rows to existing rows cryptographically. For tables that are tracked using the `BLOCKCHAIN` keyword, as rows are added, a cryptographic digest is computed for the new row plus the digest of the previous row. Any modification to the data in the chain of rows breaks the cryptographic chain. To verify a blockchain, you can either use the standard blockchain verification procedures such as `DBMS_BLOCKCHAIN_TABLE.VERIFY_ROWS` from the database, or you can independently validate the blockchain outside the database.

All historical lifespans inserted in the blockchain history table are protected using the blockchain semantics and, therefore, can be verified for any tampering. You can use blockchain functions, such as `DBMS_BLOCKCHAIN_TABLE.GET_SIGNED_BLOCKCHAIN_DIGEST` to generate a signed blockchain digest for a specified blockchain table and verify the data integrity of any row changes.

Copyright © 1999, 2025, Oracle and/or its affiliates.

For current lifespans, the relevant metadata is recorded in the `tcrv` table (`SYS_FBA_TCRV_<objno>`). You can use the `DBMS_FLASHBACK_ARCHIVE.GET_CURRENT_LIFESPAN_DIGEST` function to generate the current lifespan digest for the specified row in a tracked table. This API uses its associated metadata from the `tcrv` table to compute the current lifespan digest of the specific row of the tracked table. You can record this digest for detecting any corruptions in the row data.

Also, you can use the `DBMS_FLASHBACK_ARCHIVE.VERIFY_BLOCKCHAIN_LIFESPAN` procedure to verify current lifespans, historical lifespans, or all lifespans.

See Also

- DBMS_BLOCKCHAIN_TABLE in *Oracle Database PL/SQL Language Reference* for detailed information about the blockchain digest function and verification procedures used for blockchain tables
- DBMS_FLASHBACK_ARCHIVE in *Oracle Database PL/SQL Language Reference* for detailed information about the blockchain digest function and verification procedure for current lifespans

22.10 General Guidelines for Oracle Flashback Technology

Consider these best practices when using flashback technologies.

- Avoid issuing flashback queries on a table as of a time before Flashback Archive was enabled on the table. Alternatively, enable flashback archive before doing any DML operations on the table.
  In this example, the sequence of actions will return the row with a=1. This may look like an incorrect result, and to avoid it, you must enable flashback archive before any row is inserted.

```
CREATE TABLE foo (a NUMBER);
VAR scn1 NUMBER;
BEGIN
   SELECT DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMEBR INTO :scn FROM DUAL;
END;
INSERT INTO foo VALUES (1);
COMMIT;
ALTER TABLE foo FLASHBACK ARCHIVE;
UPDATE foo SET a=2 WHERE a=1;
COMMIT;
SELECT * FROM foo AS OF SCN :scn;
```

- 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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`, `INITRANS`, 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`.

  > **Note**
  >
  > If you cannot retrieve past data using a static data dictionary view, then you can query the corresponding base table to retrieve the data. However, Oracle does not recommend that you use the base tables directly because they are normalized and most data is stored in a cryptic format.

- You can enable optimization of data storage for history tables maintained by Flashback Archive by specifying `OPTIMIZE DATA` when creating or altering a Flashback Archive history table.

  `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.
- After you create a VPD policy, consider creating an equivalent policy for the Flashback Archive history table.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See *Oracle Database Security Guide* for more information about Virtual Private Database Policies and Flashback Time Travel.

> ⚠ **Caution**
>
> Importing user-generated history can lead to inaccurate, or unreliable results. This procedure should only be used after consulting with Oracle Support.

22.11 Oracle Virtual Private Database Policies and Oracle Flashback Time Travel

Oracle Virtual Private Database policies do not automatically work with Oracle Flashback Time Travel.

After you create an Oracle Virtual Private Database (VPD) policy for a table, consider creating an equivalent policy for the Flashback Archive history table. The following example demonstrates how to do so.

Example 22-3 Creating an Equivalent Policy for an Flashback Archive History Table

1. Create a temporary VPD administrative user.

```
CREATE USER sysadmin_vpd IDENTIFIED BY password CONTAINER = CURRENT;
GRANT CREATE SESSION, CREATE ANY CONTEXT, CREATE PROCEDURE TO sysadmin_vpd;
GRANT EXECUTE ON DBMS_SESSION TO sysadmin_vpd;
GRANT EXECUTE ON DBMS_FLASHBACK, DBMS_FLASHBACK_ARCHIVE TO sysadmin_vpd;
GRANT EXECUTE ON DBMS_RLS TO sysadmin_vpd;
GRANT UPDATE ON SCOTT.EMP TO sysadmin_vpd;
```

2. Connect to the PDB as the `sysadmin_vpd` user.

```
connect sysadmin_vpd@pdb_name
Enter password: password
Connected.
```

3. Create the VPD function.
   For example, the following function shows only rows with department number (`deptno`) 30 to users other than user `SCOTT`:

```
CREATE OR REPLACE FUNCTION emp_policy_func (
 v_schema IN VARCHAR2,
 v_objname IN VARCHAR2)

RETURN VARCHAR2 AS
condition VARCHAR2 (200);

BEGIN
 condition := 'deptno=30';
 IF sys_context('userenv', 'session_user') IN ('SCOTT') THEN
   RETURN NULL;
 ELSE
   RETURN (condition);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
 END IF;
END emp_policy_func;
/
```

4. Create the following VPD procedure to attach the `emp_policy_func` function to the `SCOTT.EMP` table.

```
BEGIN
 DBMS_RLS.ADD_POLICY (
  object_schema     => 'scott',
  object_name       => 'emp',
  policy_name       => 'emp_policy',
  function_schema   => 'sysadmin_vpd',
  policy_function   => 'emp_policy_func',
  policy_type       =>  dbms_rls.dynamic);
END;
/
```

5. Create the following `test` user and grant privileges, including those related to Flashback Archive.

```
CREATE USER test IDENTIFIED BY password;
GRANT CREATE SESSION TO test;
GRANT CONNECT, RESOURCE TO test;
GRANT SELECT ON SCOTT.EMP TO test;
GRANT FLASHBACK ARCHIVE ON ftest TO test;
GRANT EXECUTE ON DBMS_FLASHBACK_ARCHIVE TO test;
GRANT EXECUTE ON DBMS_FLASHBACK TO test;
GRANT FLASHBACK ANY TABLE TO PUBLIC;
GRANT EXECUTE ON emp_policy_func TO PUBLIC;
```

6. Enable the `SCOTT.EMP` table for flashback archive, and for transactions

```
ALTER TABLE SCOTT.EMP FLASHBACK ARCHIVE;
```

7. Perform an update to the `SCOTT.EMP` table.

```
UPDATE SCOTT.EMP SET SAL=SAL+1;
COMMIT;
```

8. Put the preceding procedure to sleep for 60 seconds.

```
EXEC DBMS_LOCK.SLEEP(60);
```

9. Connect as user `test`.

```
connect test@pdb_name
Enter password: password
Connected.
```

10. Perform the following query to show only rows that have `deptno=30`, per the VPD policy:

```
SELECT EMPNO,DEPTNO,SAL FROM SCOTT.EMP;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The VPD policy is not working because all rows are shown.

```
SELECT EMPNO,DEPTNO,SAL FROM SCOTT.EMP AS OF TIMESTAMP SYSDATE-1;
```

11. Connect as user `sysadmin_vpd`.

```
connect sysadmin_vpd@pdb_name
Enter password: password
Connected.
```

12. Find the object ID for the `EMP` table.

```
SELECT OBJECT_ID FROM DBA_OBJECTS WHERE OBJECT_NAME='EMP';
```

13. Define a similar VPD policy on the `SYS_FBA_HIST_object_id_of_EMP_table` table. This table is internally created by Flashback Archive

```
BEGIN
 DBMS_RLS.ADD_POLICY (
  object_schema     => 'scott',
  object_name       => 'sys_fba_hist_object_id_of_EMP_table',
  policy_name       => 'emp_hist_policy',
  function_schema   => 'sysadmin_vpd,
  policy_function   => 'emp_policy_func',
  policy_type       =>  dbms_rls.dynamic);
END;
/
```

14. Connect as the `test` user.

```
connect test@pdb_name
Enter password: password
Connected.
```

15. Test the policy again:

```
SELECT EMPNO,DEPTNO,SAL FROM SCOTT.EMP AS OF TIMESTAMP SYSDATE-1;
```

Now the VPD policy works, because the query only shows rows with `deptno=30`.

16. Connect as a user who can drop user accounts.
For example:

```
connect sec_admin@pdb_name
Enter password: password
Connected.
```

17. Drop the `sysadmin_vpd` user and its objects as follows:

```
DROP USER sysadmin_vpd CASCADE;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

22.12 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 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.

> **See Also**
>
> *Oracle Database Performance Tuning Guide* for information about setting the large pool size

22.13 Multitenant Container Database Restrictions for Oracle Flashback Technology

These Oracle Flashback Technology features are unavailable for a multitenant container database (CDB):

- Flashback Transaction Query is not supported in a CDB in shared undo mode. It is only supported in local undo mode.
- Flashback Transaction Backout is not supported in a CDB.

Copyright © 1999, 2025, Oracle and/or its affiliates.

23

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

23.1 Introduction to the Publish-Subscribe Model

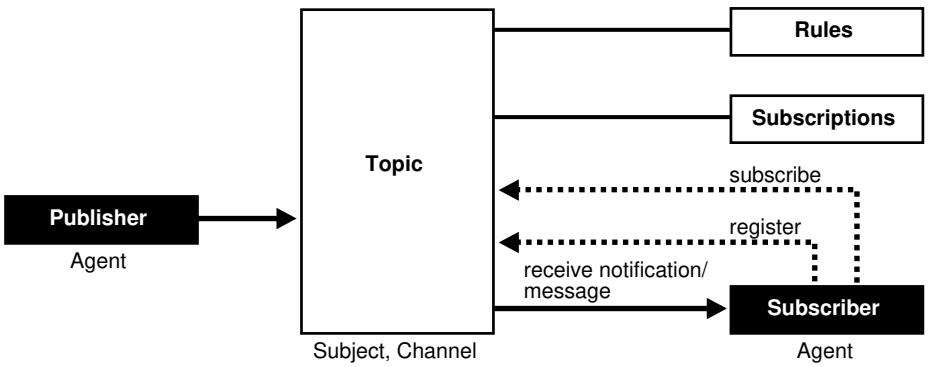
Because the database is the most significant resource of information within the enterprise, Oracle created a publish-subscribe solution for enterprise information delivery and messaging to complement this role.

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 23-1 illustrates publish and subscribe functionality.

Figure 23-1 Oracle Publish-Subscribe Functionality



Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

23.2 Publish-Subscribe Architecture

Oracle Database includes these features to support database-enabled publish-subscribe messaging:

- Database Events
- Oracle Advanced Queuing
- Client Notification

23.2.1 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*

23.2.2 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*

23.2.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

23.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

23.4 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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()
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
{
    OCISession *authp = (OCISession *) 0;
    OCISubscription *subscrhpSnoop = (OCISubscription *)0;

    /*************************************************
       Initialize OCI Process/Environment
       Initialize Server Contexts
       Connect to Server
       Set Service Context
    *************************************************/

    /* Registration Code Begins */

    /* Each call to initSubscriptionHn allocates
           and Initialises a Registration Handle */


    initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle */
        "ADMIN:PUBSUB.SNOOP", /* subscription name */
                  /* <agent_name>:<queue_name> */
        (dvoid*)notifySnoop); /* callback function */

     /*************************************************
       The Client Process does not need a live Session for Callbacks
       End Session and Detach from Server
     *************************************************/

    OCISessionEnd ( svchp,  errhp, authp, (ub4) OCI_DEFAULT);

    /* detach from server */
    OCIServerDetach( srvhp, errhp, OCI_DEFAULT);

    while (1)     /* wait for callback */
        sleep(1);

}

void initSubscriptionHn (subscrhp,
subscriptionName,
func)

OCISubscription **subscrhp;
char* subscriptionName;
dvoid * func;
{

    /* allocate subscription handle: */

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)subscrhp,
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

    /* set subscription name in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) subscriptionName,
        (ub4) strlen((char *)subscriptionName),
        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

    /* set callback function in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
        (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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

24

Using the Oracle Database ODBC Driver

For information related to the Oracle Database ODBC driver, including how to install and configure the ODBC driver, and how to use the driver to connect ODBC-enabled applications to an Oracle data source, see *Oracle Database ODBC Developer's Guide*.

Copyright © 1999, 2025, Oracle and/or its affiliates.

25

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:

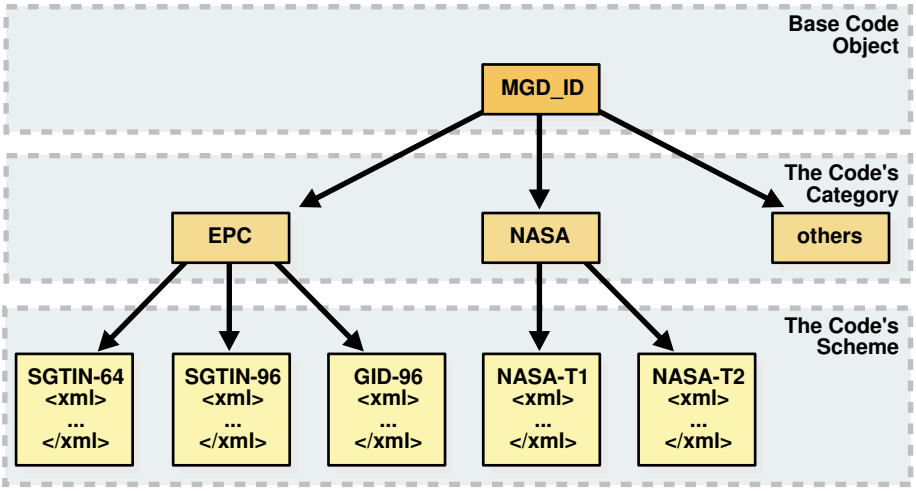
- Identity Concepts
- What is the Identity Code Package?
- Using the Identity Code Package
- Identity Code Package Types
- DBMS_MGD_ID_UTL Package
- Identity Code Metadata Tables and Views
- Electronic Product Code (EPC) Concepts
- Oracle Database Tag Data Translation Schema

25.1 Identity Concepts

A database object `MGD_ID` is defined that lets users use EPC standard identity codes and use their own existing identity codes. 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 25-1 and load these into the database as well.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 25-1 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 25-1. 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 25-1 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)
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)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 25-1 (Cont.) General Structure of EPC Encodings

Encoding Name	Header Length in bits	Field Names (parameter_list name-value pairs) and (length in bits)
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: 10011000000000000010000011101100010000100000111111100110001100 10
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

Copyright © 1999, 2025, Oracle and/or its affiliates.

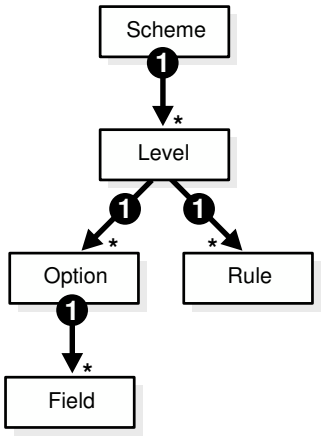
compatible scheme and seamlessly translate it into the generic Oracle Database defined metadata. See the `EPC_TO_ORACLE` Function in Table 25-4 for more information.

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 25-3 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. Developers can refer to this Oracle Database TDT XML schema to define their own tag structures.

Figure 25-2 shows the Oracle Database Tag Data Translation Markup Language Schema diagram.

Figure 25-2 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.

See Also

- See Electronic Product Code (EPC) Concepts for a brief description of EPC concepts
- See Oracle Database Tag Data Translation Schema for the actual Oracle Database TDT XML schema

Copyright © 1999, 2025, Oracle and/or its affiliates.

25.2 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 25-2 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.

  Using the Identity Code Package describes how to use these ADTs and member functions.

  Identity Code Package Types and DBMS_MGD_ID_UTL Package briefly describe the reference information for these ADTs along with a set of utility subprograms.
- `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:

- Electronic Product Code (EPC) Concepts describes the EPC spec v1.1 product code and its family of coding schemes.
- Identity Code Metadata Tables and Views 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:

- Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type 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:

- Identity Code Package Types and DBMS_MGD_ID_UTL Package 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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, `catagory_id` attribute values, and components structures for your own encoding types. See these topics for more information:

- Creating a Category of Identity Codes describes how to create your own identity codes by first registering the encoding category, and then registering the schemes associated to the encoding category.
- Identity Code Metadata Tables and Views 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.

> **See Also**
>
> See *Oracle Database PL/SQL Packages and Types Reference* for detailed reference information.

25.3 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

25.3.1 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

25.3.1.1 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);
             ...);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL*Plus command:

```
describe warehouse_info;
```

Result:

```
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
CODE                                      NOT NULL MGDSYS.MGD_ID
ARRIVAL_TIME                                       TIMESTAMP(6)
LOCATION                                           VARCHAR2(256)
```

25.3.1.2 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

25.3.1.2.1 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('example.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')))
.
.
.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

25.3.1.2.2 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('example.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('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')))
.
.
.
```

25.3.1.2.3 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('example.com', '80');
call DBMS_MGD_ID_UTL.refresh_category
  (DBMS_MGD_ID_UTL.get_category_id('EPC', NULL));
select MGD_ID('EPC', NULL,
              MGD_ID_COMPONENT_VARRAY(
              MGD_ID_COMPONENT('companyprefix','0037000'),
              MGD_ID_COMPONENT('itemref','030241'),
              MGD_ID_COMPONENT('serial','1041970'),
              MGD_ID_COMPONENT('schemes','SGTIN-64')
             )
            ) from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

@constructor33.sql
.
.
.
MGD_ID('1', MGD_ID_COMPONENT_VARRAY
             (MGD_ID_COMPONENT('companyprefix', '0037000'),
              MGD_ID_COMPONENT('itemref', '030241'),
              MGD_ID_COMPONENT('serial', '1041970'),
              MGD_ID_COMPONENT('schemes', 'SGTIN-64')
             )
       )
.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
.
.
```

25.3.1.2.4 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('example.com', '80');
call DBMS_MGD_ID_UTL.refresh_category
  (DBMS_MGD_ID_UTL.get_category_id('EPC', NULL));
select MGD_ID('EPC', NULL,
              'urn:epc:id:sgtin:0037000.030241.1041970',
              'filter=3;scheme=SGTIN-64') from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

@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')
       )
      )
.
.
.
```

25.3.1.3 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('example.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');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
INSERT INTO WAREHOUSE_INFO (code, arrival_time, location)
  values (MGDSYS.MGD_ID ('EPC',
                         NULL,
                         'urn:epc:id:sgtin:0037000.020140.10174832',
                         null
                        ),
          SYSDATE,
          'SHELF_1034');

COMMITT;
call DBMS_MGD_ID_UTL.remove_proxy();
```

25.3.1.4 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 Using the get_component Function with the MGD_ID Object for more information and see Table 25-3 for a list of member functions.

25.3.2 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'));
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

25.3.3 Using MGD_ID ADT Functions

The MGD_ID ADT contains member subprograms that operate on these ADTs. See Table 25-2 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

25.3.3.1 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.

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 25-3 for a list of other member functions.

> **See Also**
>
> Defining a Category of Identity Codes and Adding Encoding Schemes to an Existing Category for more information about how to create a identity code type

25.3.3.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 25-1 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 Identity Concepts, 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,
    '10011000000000000010000011101100010000100000111111100110001100l0',
    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'),
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
        MGD_ID_COMPONENT('companyprefix', '0037000'),
        MGD_ID_COMPONENT('serial', '1041970'),
        MGD_ID_COMPONENT('itemref', '030241')
      )
    )
```

25.3.3.3 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
```

25.3.3.4 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
--------------------------------------------------------------------------------
10011000000000000010000011101100010000100000111111100110001100 10
```

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.

25.3.4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

25.3.4.1 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 Adding Two Metadata Schemes to a Newly Created Category for an example.

25.3.4.2 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">
 <scheme name="CONTRACTOR_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.contractor.">
   <option optionKey="1" pattern="mycompany.contractor.([0-9]*).([0-9]*)"
           grammar="''mycompany.contractor.'' contractorID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="contractorID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="11">
   <option optionKey="1" pattern="11([01]{7})([01]{6})"
           grammar="''11'' contractorID divisionID ">
    <field seq="1" characterSet="[01]*" name="contractorID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>
```

The `CONTRACTOR_TAG` scheme contains two encoding levels, or ways in which the tag can be represented. The first level is `URI` and the second level is `BINARY`. The `URI` representation starts with the prefix "`mycompany.contractor`." and is then followed by two numeric fields separated by a period. The names of the two fields are `contractorID` and `divisionID`. The pattern field in the option tag defines the parsing structure of the tag `URI` representation, and the grammar field defines how to reconstruct the URI representation. The `BINARY` representation can be understood in a similar fashion. This representation starts with the prefix "01" and is then followed by the same two fields, `contractorID` and `divisionID`, this time, in their respective binary formats. Given this XML metadata structure, contractor tags can now be decoded from their `URI` and `BINARY` representations and the resulting fields can be re-encoded into one of these representations.

The `EMPLOYEE_TAG` scheme is defined in a similar fashion and is shown as follows.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                <?xml version="1.0" encoding="UTF-8"?>
                <TagDataTranslation version="0.04" date="2005-04-18T16:05:00Z"
                                    xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                                    xmlns="oracle.mgd.idcode">
                 <scheme name="EMPLOYEE_TAG" optionKey="1" xmlns="">
                  <level type="URI" prefixMatch="mycompany.employee.">
                   <option optionKey="1" pattern="mycompany.employee.([0-9]*).([0-9]*)"
                           grammar="''mycompany.employee.'' employeeID ''.'' divisionID">
                    <field seq="1" characterSet="[0-9]*" name="employeeID"/>
                    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
                   </option>
                  </level>
                  <level type="BINARY" prefixMatch="01">
                   <option optionKey="1" pattern="01([01]{7})([01]{6})"
                           grammar="''01'' employeeID divisionID ">
                    <field seq="1" characterSet="[01]*" name="employeeID"/>
                    <field seq="2" characterSet="[01]*" name="divisionID"/>
                   </option>
                  </level>
                 </scheme>
                </TagDataTranslation>;
```

To add these schemes to the category ID previously created, use the `DBMS_MGD_ID_UTIL.add_scheme` function.

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('example.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">
 <scheme name="CONTRACTOR_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.contractor.">
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   <option optionKey="1" pattern="mycompany.contractor.([0-9]*).([0-9]*)"
           grammar="''mycompany.contractor.'' contractorID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="contractorID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="11">
   <option optionKey="1" pattern="11([01]{7})([01]{6})"
           grammar="''11'' contractorID divisionID ">
    <field seq="1" characterSet="[01]*" name="contractorID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>';

  amt := length(buf);
  pos := 1;
  DBMS_LOB.WRITE(tdt_xml, amt, pos, buf);
  DBMS_LOB.CLOSE(tdt_xml);

  DBMS_MGD_ID_UTL.ADD_SCHEME(category_id, tdt_xml);

  -- add employee scheme to the category
  DBMS_LOB.CREATETEMPORARY(tdt_xml, true);
  DBMS_LOB.OPEN(tdt_xml, DBMS_LOB.LOB_READWRITE);

  buf := '<?xml version="1.0" encoding="UTF-8"?>
<TagDataTranslation version="0.04" date="2005-04-18T16:05:00Z"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                    xmlns="oracle.mgd.idcode">
 <scheme name="EMPLOYEE_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.employee.">
   <option optionKey="1" pattern="mycompany.employee.([0-9]*).([0-9]*)"
           grammar="''mycompany.employee.'' employeeID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="employeeID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="01">
   <option optionKey="1" pattern="01([01]{7})([01]{6})"
           grammar="''01'' employeeID divisionID ">
    <field seq="1" characterSet="[01]*" name="employeeID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>';

  amt := length(buf);
  pos := 1;
  DBMS_LOB.WRITE(tdt_xml, amt, pos, buf);
  DBMS_LOB.CLOSE(tdt_xml);
  DBMS_MGD_ID_UTL.ADD_SCHEME(category_id, tdt_xml);

  -- validate the scheme
  dbms_output.put_line('Validate the MGD_SAMPLE_CATEGORY Scheme');
  validate_tdtxml := DBMS_MGD_ID_UTL.validate_scheme(tdt_xml);
  dbms_output.put_line(validate_tdtxml);
  dbms_output.put_line('Length of scheme xml is: '||DBMS_LOB.GETLENGTH(tdt_xml));

  -- test tag translation of contractor scheme
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
.
.
.
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
.
.
.
```

25.4 Identity Code Package Types

Table 25-2 describes the Identity Code Package ADTs.

Table 25-2 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 25-3 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 25-3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 25-3 (Cont.) MGD_ID ADT Subprograms

Subprogram	Description
TRANSLATE Static Function	Translates one MGD_ID representation of an identity code into a different MGD_ID representation.

25.5 DBMS_MGD_ID_UTL Package

Table 25-4 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 25-4 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.
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_CATEORY Function	Removes a category including all the related TDT XML.
REMOVE_PROXY Procedure	Unsets the host and port of the proxy server.
REMOVE_SCHEME Procedure	Removes the tag scheme for a category.
SET_JAVA_LOGGING_LEVEL Procedure	Sets the Java logging level.
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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 25-4 (Cont.) DBMS_MGD_ID_UTL Package Utility Subprograms

Subprogram	Description
`VALIDATE_SCHEME` Function	Validates the input tag data translation XML against the Oracle Database tag data translation schema.

25.6 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Metadata View Definitions

Table 25-5, Table 25-6, Table 25-7, and Table 25-8 describe the metadata view definitions for the MGD_ID_CATEGORY, USER_ID_CATEGORY, MGD_ID_SCHME, and USER_MGD_ID_SCHME respectively as defined in the mgdview.sql file.

Table 25-5 Definition and Description of the MGD_ID_CATEGORY Metadata View

Column Name	Data Type	Description
CATEGORY_ID	NUMBER(4)	Category identifier
CATEGORY_NAME	VARCHAR2(256)	Category name
AGENCY	VARCHAR2(256)	Organization that defined the category
VERSION	VARCHAR2(256)	Category version
URI	VARCHAR2(256)	URI that describes the category

Table 25-6 Definition and Description of the USER_MGD_ID_CATEGORY Metadata View

Column Name	Data Type	Description
CATEGORY_ID	NUMBER(4)	Category identifier
CATEGORY_NAME	VARCHAR2(256)	Category name
AGENCY	VARCHAR2(256)	Organization that defined the category
VERSION	VARCHAR2(256)	Category version
URI	VARCHAR2(256)	URI that describes the category

Table 25-7 Definition and Description of the MGD_ID_SCHEME Metadata View

Column Name	Data Type	Description
CATEGORY_ID	NUMBER(4)	Category identifier
TYPE_NAME	VARCHAR2(256)	Encoding scheme name, for example, SGTIN-96, GID-96, and so 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)

Table 25-8 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 25-8 (Cont.) Definition and Description of the USER_MGD_ID_SCHEME Metadata View

Column Name	Data Type	Description
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)

25.7 Electronic Product Code (EPC) Concepts

Topics:

- RFID Technology and EPC v1.1 Coding Schemes
- Product Code Concepts and Their Current Use

25.7.1 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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

25.7.2 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

25.7.2.1 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 EPCGlobal EPC Data Standards Version 1.1 defines the abstract content of the Electronic Product Code, and its concrete realization in the form of RFID tags, Internet URIs, and other representations. In EPC Version 1.1, the specific coding schemes include a General Identifier (GID), a serialized version of the EAN.UCC Global Trade Item Number (GTIN®), the EAN.UCC Serial Shipping Container Code (SSCC®), the EAN.UCC Global Location Number (GLN®), the EAN.UCC Global Returnable Asset Identifier (GRAI®), and the EAN.UCC Global Individual Asset Identifier (GIAI®).

Copyright © 1999, 2025, Oracle and/or its affiliates.

25.7.2.1.1 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.

25.7.2.1.2 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.

25.7.2.1.3 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.

25.7.2.1.4 EPC Identity URI

The EPC identity URI is a representation of a pure identity as a Uniform Resource Identifier (URI).

25.7.2.1.5 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`.

25.7.2.1.6 EPC Encoding Procedure

The EPC encoding procedure generates an EPC tag bit-level encoding using various information.

25.7.2.1.7 EPC Decoding Procedure

The EPC decoding procedure converts an EPC tag bit-level encoding to an EAN.UCC code.

25.7.2.2 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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

25.7.2.3 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.

25.7.2.4 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.

25.7.2.5 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.

25.7.2.6 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.

25.7.2.7 RFID EPC Network

The RFID EPC network identifies, tracks, and locates assets. Physical objects are identified by a unique RFID enabled EPC.

25.8 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>
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
<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"/>
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  <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>
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

26
Microservices Architecture

This chapter explains the microservices architecture and the benefits of using it to build your microservices-based database applications.

Topics:

- About Microservices Architecture
- Features of Microservices Architecture
- Challenges in a Distributed System
- Solutions for Microservices

26.1 About Microservices Architecture

A microservices architecture is an approach to developing applications as a collection of loosely coupled, autonomous services. A microservice in an application is a small, self-contained service with a limited contract. For example, a travel agency can implement a microservices application that provides airline, hotel, and car rental bookings as microservices.

Each service in a microservices application implements a single business capability and communicates with other services through APIs or a messaging system. With loosely coupled services and encapsulated data, you can independently scale individual components, have a dedicated team for each service, and build your applications to achieve greater business agility and profits.

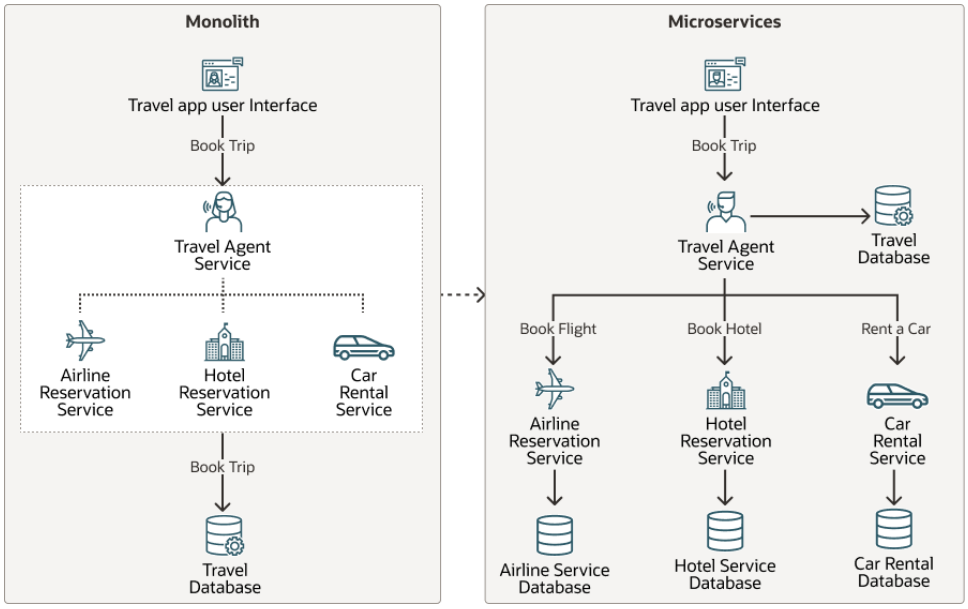
Why switch from a monolithic application to microservices?

- Increased complexity of the application architecture is making your application difficult to scale and manage.
- Changing or adding a single feature in a monolith requires you to change the code for the entire application, making it a time-consuming and expensive process.
- Your product is mature enough to scale up and your team has the necessary tools and skills to adopt microservices.

Monolith and Microservices illustrates a travel agency application as a monolithic application and as a micorservices-based application with flight reservation, hotel booking, and car rental services.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 26-1 Monolith and Microservices



Benefits of Microservices

- Individual application components can scale independently. Applications operating within a cloud or hybrid environment can scale easily with independent development and deployment of services.
- Applications are easier to build, maintain, and deploy. Each service can be built and deployed independently, without affecting other services or rebuilding the entire application.
- Software development is parallelized. You have a team working on a separate codebase, making it easy to develop and test software, and adopt new technology.
- Applications can be built in multiple languages, like Java, .NET, or Python, and using various data types, such as JSON, Graph, and Spatial. Each service team can choose its own technology stack and tools because microservices communicate through language-agnostic APIs.
- Data ownership is decentralized. Each service has its own database, or there might be one database server but within that server each service has its own private schema and tables.

See Also

- Learn About the Microservices Architecture (oracle.com)
- Differences Between Microservices and Monolithic Architectures

Copyright © 1999, 2025, Oracle and/or its affiliates.

26.2 Features of Microservices Architecture

A microservices architecture must have these features:

Loosely-coupled Services

Microservices architecture requires breaking down your application into smaller component services. You can deploy each component service individually. Each self-contained service implements a core function and has clearly defined boundaries with data divided into bounded contexts. The database schema is restructured to identify which datasets each service needs.

Decentralized Data

Microservices ensure autonomy of services and teams. Services may not share the same data source or technology. For example, you could think of user registration and billing management as different teams, skilled in their respective areas, and working on a technology stack that is suited to their specific requirements and skillsets.

Independent Deployment

Independent services are developed and deployed in small, manageable units without affecting a large part of the codebase.

Automated Infrastructure

Microservices use automated infrastructure to operate and may require:

- An interservice communication system using asynchronous messaging and message brokers.
- A containerized system that allows you to focus on developing the services and lets the system handle the deployment and dependencies.

Reusable Systems

Developers can reuse code or reuse library functions (if using the same language and platform) in another feature or across services and teams.

Resilient Architecture

Microservices can withstand an entire application crash because if an independent service fails, you can expect to lose some part of the application functionality but other parts of the application can continue functioning.

26.3 Challenges in a Distributed System

To ensure data integrity and consistency, a transaction must have ACID (Atomicity, Consistency, Isolation, Durability) properties. In a monolith, we have a database system to ensure ACIDity because a local database transaction works on a single database system. A transaction has either all steps complete or no steps complete. If any step fails, the transaction is rolled back.

When designing a microservices-based architecture, you need to be aware of its challenges. Here we look at some common challenges.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Service Decomposition

When designing microservices, you must ensure that your monolith or legacy application is split into loosely coupled components with clearly defined boundaries. You need to check for the dependencies between components to see if they are sufficiently independent.

Complexity

A microservices application, being a distributed system, can have business transactions that span multiple systems and services. Service-level transactions (for clarity, let's call service-level transactions "sub-transactions") are called in a sequence or in parallel to complete the entire transaction. You must deal with additional complexity of creating a distributed system that uses an interservice communication mechanism and is designed to handle partial failures.

Distributed Transactions

In a microservices architecture, a monolithic system is decomposed into self-encapsulated services. With a database per microservice approach, to ensure that a transaction is complete, the transaction must span across multiple databases. If one of the sub-transactions fails, you must roll back the successful transactions that were previously completed. To maintain transaction atomicity, services need interservice communication and coordination to ensure that all the services commit on success of the transaction or the services that commit, roll back on failure of the transaction.

Transaction Consistency and Isolation

Another challenge involves handling concurrent requests. A sub-transaction works by keeping its data source (rows) locked until the result of the transaction is known. If multiple service calls try to simultaneously access the same data source, you must have a mechanism to handle concurrency and determine the amount of data that must be made available to concurrent transactions. In a Long-running Action (LRA) like a bank transaction that spans multiple services, data can be held in a locked state at multiple objects for the transaction lifecycle. To ensure that data remains consistent across services, you must implement a concurrency control mechanism using data locks.

Interservice Communication

In a monolith, application components use function calls to invoke each other. Microservices interact with each other over the network. Interservice communication using asynchronous messaging is essential when propagating changes across multiple microservices.

26.4 Solutions for Microservices

Two patterns, namely Two-phase Commit (2PC) and Saga, attempt to alleviate the challenges and bring in consistency in the distributed transaction management. 2PC is ideal for immediate transactions, and Saga works well if you wish to implement distributed transactions for long running action (LRA).

For enterprises looking for microservices solutions, Oracle has developed 12 proven patterns that are crucial for microservices success. You can find more information about these 12 patterns in the following section.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Backend as a Service For The 12 Patterns For Microservices Success

26.4.1 Two-Phase Commit Pattern

Two-phase commit (2PC) is an atomic commit protocol where you have a coordinator that unilaterally decides the outcome of the transaction. All participants in a transaction must commit or roll back based on the coordinator's decision. All processes that occur in two or more tables, such as insert, update, delete, or all, commit or roll back simultaneously.

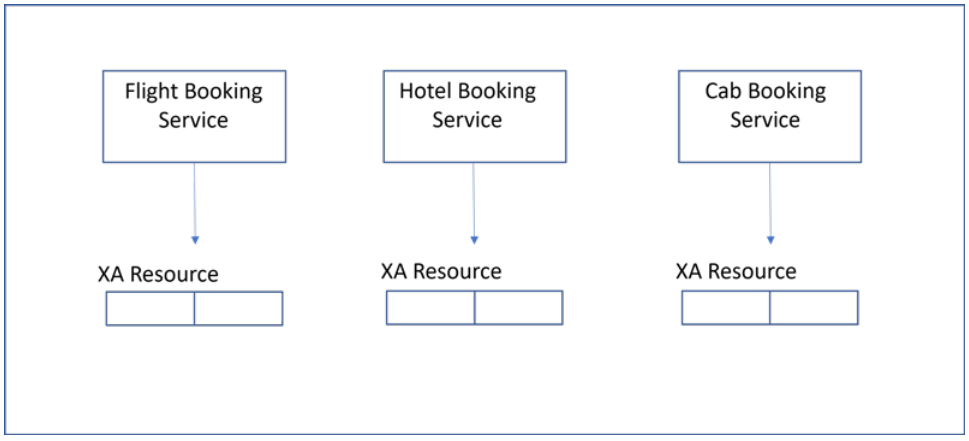
2PC has two phases: A prepare phase and a commit phase. In the prepare phase, the transaction coordinator sends a prepare command to each microservice. Each microservice checks if they can guarantee that they can commit the transaction and if yes, they send back a “prepared” response to the transaction coordinator.

After all the microservices are prepared, the coordinator directs the microservices to commit the change. If any microservice does not respond to the prepare command or sends back a failed response, the transaction coordinator sends a cancel command to all microservices. The sub-transactions are rolled back, canceling all the changes.

To ensure that a microservice can commit, 2PC ensures that the locks on modified data and the decision to prepare are in a durable storage. This lock remains active until the commit or rollback, and no other transaction can use this information. These locks can become bottlenecks that slow down your system. In an application handling multiple services, 2PC can create complexity and adverse performance impacts.

Using 2PC would reserve the flight, hotel, and cab services at the same time as shown in the Two-Phase Commit illustration below. If the transaction fails, none of the services are booked.

Figure 26-2 Two-Phase Commit



> **See Also**
>
> Two-Phase Commit for more information about 2PC pattern.

Copyright © 1999, 2025, Oracle and/or its affiliates.

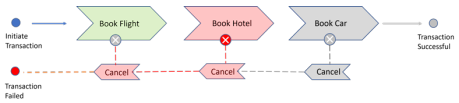
26.4.2 Saga Design Pattern

The Saga design pattern provides transaction management for microservices and enables business transactions to maintain data consistency across microservices. A Saga breaks up a distributed transaction into a sequence of independent local transactions (sub-transactions). Each service performs its own sub-transaction and publishes an event or message. The successive services listen to that event or message and perform the next sub-transaction.

A transaction spans multiple databases (PDBs) and may perform a sub-transaction for the database in each microservice. A business transaction is successful if all sub-transactions in the sequence complete successfully. If any of the sub-transactions fails, compensating transactions are invoked to “undo” the state in the database(s) affected by the changes of the preceding sub-transactions. Each sub-transaction can have ACID properties on a single database. Each sub-transaction in a Saga has a corresponding, compensating transaction that is executed if there is a rollback.

The Saga Transaction below illustrates the sequence of a Saga transaction when the transaction is successful and when the transaction fails prompting a rollback.

Figure 26-3 Saga Transaction



An important facet of the Saga pattern is the Saga coordinator. The Saga coordinator tells other participants what to do. The coordinator invokes Saga participants and with every response, the coordinator transitions to the next state. Asynchronous messaging is another important aspect of Sagas, which involves sending interservice messages over a queuing system.

The Saga pattern enables you to build more robust systems because Sagas use a failure management pattern. Every action has a compensating action for rollback, which helps ensure eventual data consistency and correctness across microservices.

26.4.2.1 Why Use Sagas?

Use Sagas for the following reasons:

- Perform a group of operations related to different microservices automatically.
- Rely on the Saga pattern to ensure data consistency across microservices and in different databases.
- Reduce the locking period and restrict the data locks to the duration of local transactions for improved concurrency.
- Avoid using Two-phase Commit (2PC) because of its extended locking period and performance constraints.
- Roll back or compensate if one of the transaction operations in the sequence of microservices fails.
- Use microservices that do not support 2PC.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Why using Sagas is a better option than using 2PC transactions?

- A 2PC transaction can cause extended locking period and incomplete results for queries until the 2PC transaction is committed or canceled. With Sagas, the data locks are placed only for the duration of the local transaction (that implements a microservice transaction), and not for the entire Saga lifecycle. Reducing the locking period improves the throughput of Saga transactions, resulting in better scalability for applications.
- Sagas are useful for long-lived transactions. 2PC is not ideal for long-lived transactions since the locking of resources for prolonged durations can affect performance and scalability.

When you use Sagas with Oracle Database, lock-free reservation features that are built into the database help improve concurrency and reduce bottlenecks.

26.4.2.2 Saga Implementation Approaches

There are two common saga implementation approaches, namely the orchestration and choreography models.

Orchestration

All communication between microservices is made through a centralized service called a Saga coordinator. The coordinator service is responsible for receiving the requests and calling the respective services. If any service fails, the coordinator service implements the roll back methods. You can use the orchestration model for complex workflows that need the Saga coordinator services.

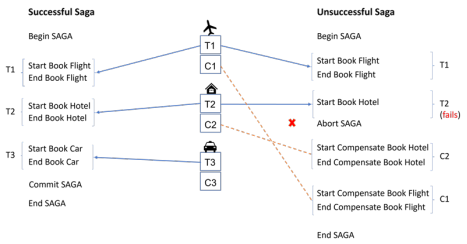
Choreography

In the choreography model, services communicate amongst each other and if any request fails, each service must have its own fallback method to roll back the transaction. If you have simpler workflows, you can use the choreography model.

26.4.2.3 Successful and Unsuccessful Sagas

Successful and Unsuccessful Sagas illustrates a successful and an unsuccessful Saga transaction.

Figure 26-4 Successful and Unsuccessful Sagas



26.4.2.4 Saga Flow

Taking the example of the travel agency application, let us look at a trip booking Saga flow.

1. A travel agency application user sends a book trip request from the application UI, which is sent to the travel agency service.

Copyright © 1999, 2025, Oracle and/or its affiliates.

2. The travel agency service calls on the Saga coordinator to begin the Saga.
3. The coordinator includes a Saga identifier (Saga ID) in its response.
4. The travel agency registers itself with the Saga. The agency sends the Saga completion (compensation or commit) callback to the coordinator. Each participant in a Saga must provide the Saga with a callback that is used at the time of Saga completion.
5. The travel agency adds the Saga ID to a request call to the flight microservice to book a flight.
6. The participant microservice (flight) contacts the coordinator to join the Saga (register itself to the Saga). The participant service also sends its Saga completion (compensation or commit) callback to the coordinator.
7. The travel agency repeats the same process for other participants, such as the hotel and car rental services.
8. The participant microservices execute the business process.
9. After the travel agency receives all replies, it determines whether to commit or roll back the Saga and informs the coordinator.
10. The coordinator calls the appropriate callbacks (compensation or commit) for the Saga participants and returns the control to the travel agency.
11. The travel agency confirms the success or failure of the Saga to the coordinator and ends the Saga.

26.4.3 Backend as a Service For The 12 Patterns For Microservices Success

If you search for microservices success stories, you find that the industry is divided on the advantages that microservices can bring to businesses.

Traditionally, monoliths have been the standard architecture where the entire application has a single domain and data access pattern, keeping the transactional boundaries local within a single database; and rarely do applications need a distributed transaction. This brings in some simplicity to data design, but it also results in a spaghetti pattern of accesses to the tables in a schema, and a hugely complex entity-relationship setup. Therefore, one big disadvantage of monoliths is the complexity of making changes and the time it takes to launch a new feature in the application.

Microservices (microservices architectures), on the other hand, advertises its advantages as agility, which it achieves by bounding the context and using loose coupling between the various microservices. This limits the access patterns (to the data from a microservice) to be local, and in very rare cases, rely on the transactions across microservices, supported by Sagas, which are asynchronous and scalable.

However, some of the advantages of microservices are negated by (1) the overhead of setting up the infrastructure for integration of microservices with APIs or messaging, and (2) the overhead of testing even a single microservice change when the number of microservices is in the hundreds. Bounded contexts are important to identify such data layouts that force loose coupling of data accesses across microservices, moving away from the tight coupling used in monoliths. Extracting microservices from existing monoliths is also possible using the Strangler pattern, where independent functions can be determined by doing a bottoms-up affinity analysis. However, due to such issues, most enterprises struggle to build and deploy microservices architectures.

For enterprises using microservices, Oracle has leveraged its vast experience to develop the following 12 patterns for microservices success:

Copyright © 1999, 2025, Oracle and/or its affiliates.

1. **Bounded contexts** - design it upfront, or break monoliths into microservices with a data refactoring advisor and

   Loose coupling - decouple data by isolating schema to the microservice, and using a reliable event mesh
2. **Transactional Outbox** - sending a message and a data manipulation operation in a single local transaction
3. **Reliable Event Mesh** - Event Mesh for all events – with high throughput transactional messaging and pub/sub; with event transformations and event routing, alongside Kafka (if needed)
4. **Sagas** - transactions across microservices with support from Event Mesh and Escrow journaling in the database
5. **Security** for microservices - securing each endpoint from an API gateway, to load balancer, to event mesh, to the database
6. **Polyglot microservices** - support for microservices and message formats in a variety of languages using JSON as the payload
7. **Unified observability** for microservices - metrics, logs, and traces into a single dashboard for tuning and self-healing
8. **Event Aggregation** - events are ephemeral and notify/trigger real-time action; after which they are aggregated into the database – the database is the ultimate compacted topic
9. **CQRS** - Command Query Responsibility Segregation, where operational and analytical copies of data are available for microservices
10. **CI/CD** for microservices - a multi-tenant database makes it a natural fit for microservice + PDB to deploy/build independently; and also an app (with Jenkins or GitHub Actions) and schema (Liquibase, Flyway, and EBR in the database)
11. **AI Sandbox for Microservices** - an experimental sandbox to run RAG and related techniques to increase enterprise accuracy for Chatbots and other use cases – making them ready for deployment
12. **Backend as a Service (BaaS)** - a microservices infrastructure right sized for test/dev and production deployments (small, medium) on any cloud or on-premises deployments; for Spring Boot Apps

The Backend as a Self-Service (BaaS) pattern brings all the preceding patterns into a successful dev/test and production environment by making the deployment of the microservices platform easy.

The next chapter discusses about Oracle Backend as a Self-Service (OBaaS), which is also called the Oracle Backend for Microservices and AI platform.

Spring Boot is the most popular microservices framework for Java, and OBaaS enables Java developers to simplify the task of building, testing, and operating Spring Boot-based microservices.

Copyright © 1999, 2025, Oracle and/or its affiliates.

27

Oracle Backend for Microservices and AI

This chapter explains the Oracle Backend for Microservices and AI platform (also referred to as OBaaS).

Topics:

- About Oracle Backend for Microservices and AI
- Getting Started with Oracle Backend for Microservices and AI
- CloudBank Sample Application

27.1 About Oracle Backend for Microservices and AI

Spring Boot is a popular framework for building modern, cloud-native applications. 80% of the modern enterprise application development in Java uses Spring Boot for microservices. As noted earlier, deploying microservices in the production environment is not an easy task. A Backend as a Self-Service (BaaS) is the most important pattern for success because it executes the other 11 patterns (discussed earlier) in an integrated platform, which can be deployed in a few clicks, on any cloud – public or private.

The Oracle Backend for Microservices and AI platform, which is also referred to as Oracle Backend as a Self-Service (OBaaS), includes these important features:

- OBaaS deploys with Kubernetes on multiple clouds, on-premises, and on a laptop for rapid development of microservices using Spring Starters for Oracle Database.
- Applications can take advantage of the converged Oracle Database that features a variety of data types – relational, document/JSON, graph, spatial – and serves both the data at rest and data in motion use cases for the most challenging enterprise applications.
- Oracle Database includes an event queue or a streaming feature called Transactional Event Queues (TxEventQ) that is very similar to Apache Kafka for streaming use cases, making Oracle Database a message broker.
- Containerization of Oracle Database with pluggable databases (PDBs) makes it a natural fit for isolating microservices to a single PDB, separated by bounded contexts for each microservice, or in some cases, using schema-level isolation with fewer PDBs to contain the costs of deployment.

27.2 Getting Started with Oracle Backend for Microservices and AI

This section provides essential information for developers looking to get started with the Oracle Backend for Microservices and AI platform.

The Oracle Backend for Microservices and AI platform (OBaaS) is described at the following location: https://bit.ly/OracleAI-microservices, which also includes information about using the deployment paths on Oracle Cloud Infrastructure (OCI) from the OCI Marketplace to deploy the platform in a few clicks using the OCI Resource Manager templates with Terraform and Ansible. Additionally, using the custom install path, you can deploy the platform on Microsoft

Copyright © 1999, 2025, Oracle and/or its affiliates.

Azure, OpenShift, Tanzu – allowing Bring Your Own (BYO) of Oracle Database, Kubernetes cluster, and networking.

In addition to an Oracle Autonomous Database Serverless instance, the following software components are deployed in an OCI Container Engine for Kubernetes cluster (OKE cluster). The following list of components is an example of the services that run in OBaaS for supporting application deployment in the production environment.

> **Note**
>
> Always check for the latest release of these software components at the following location: https://bit.ly/OracleAI-microservices.

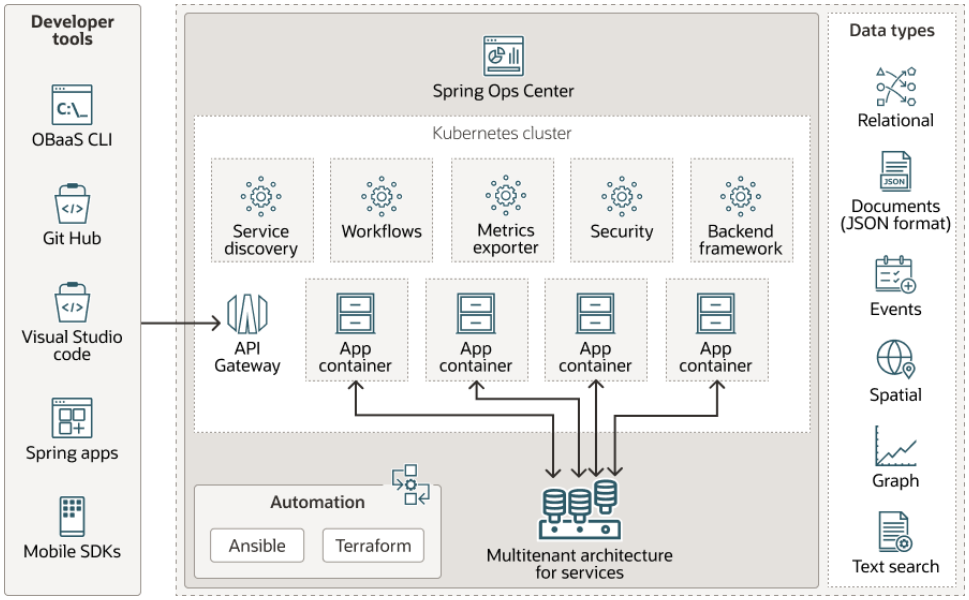
- Apache APISIX API Gateway and Dashboard
- Apache Kafka
- Coherence Operator
- Conductor Server
- Grafana
- HashiCorp Vault
- Jaeger
- Apache Kafka
- Loki
- Netflix Conductor
- OpenTelemetry Collector
- Oracle Autonomous Database Serverless
- Oracle Backend for Spring Boot Command Line Interface (CLI)
- Oracle Backend for Spring Boot Visual Studio Code plug-in
- Oracle Database Operator for Kubernetes (OraOperator or the operator)
- Oracle Transaction Manager for Microservices (MicroTx)
- Parse and Parse Dashboard (optional)
- Prometheus
- Promtail
- Spring Boot Admin dashboard
- Spring Cloud Config server
- Spring Cloud Eureka service registry
- Strimzi Kafka Operator

OBaaS integrates multiple platform services that are needed for applications and data; using Oracle Database with Kubernetes and the Oracle Database Operator for Kubernetes, thus making DevOps for microservices simple.

The following figure illustrates a high-level view of the OBaaS platform's architecture.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 27-1 Oracle Backend for Microservices and AI (OBaaS)



Developers also have access to development or build-time services and libraries, including:

- A command-line interface (CLI) to manage service deployment and configuration, including database schema management
- Visual Studio Code (VS Code) plug-in to manage service deployment and configuration
- Spring Data (Java Persistence API (JPA) and Oracle JDBC) to access Oracle Database
- Oracle Java Database Connectivity (Oracle JDBC) Drivers
- Spring Cloud Config Client
- Spring Eureka Service Discovery Client
- Spring Cloud OpenFeign
- OpenTelemetry Collector (including automatic instrumentation)
- Spring Starters for Oracle Universal Connection Pool (UCP), Oracle Wallet, Oracle Advanced Queuing (AQ), and Transactional Event Queues (TxEventQ)

OBaaS is also extensible, and so you can add additional Kubernetes Operators (example, for Weblogic) and other components that they deem important (example, Redis). These additional components may get included in OBaaS in the future; if there are enough customers requesting a tested platform, and a path to support from the community (if open source), or from a vendor (if using a free or paid version).

For more information about the OBaaS deployment and how to use OBaaS in the production environment, see https://bit.ly/OracleAI-microservices.

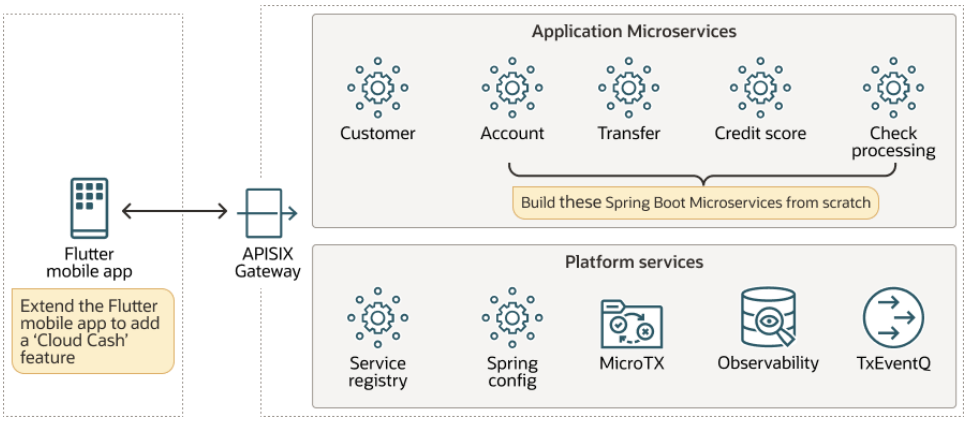
27.2.1 CloudBank Sample Application

Oracle provides a sample application called CloudBank that can be used to learn how to use the Oracle Backend for Microservices and AI platform (OBaaS).

Copyright © 1999, 2025, Oracle and/or its affiliates.

CloudBank is a microservices-based application that installs the Oracle Backend for Microservices and AI platform and deploys a few microservices – Customer, Account, Transfer, Credit Score, Check Processing – as shown in the following figure.

Figure 27-2 CloudBank Sample Application



In the CloudBank hands-on lab, you can learn how to:

- Install Oracle Backend for Microservices and AI.
- Set up a development environment for Spring Boot.
- Build Spring Boot microservices from scratch using Spring Web to create Representational State Transfer (REST) services.
- Use service discovery and client-side load balancing.
- Use Spring Actuator to allow monitoring of services.
- Create services that use asynchronous messaging with Java Message Service (JMS) instead of REST.
- Implement the Saga pattern to manage data consistency across microservices.
- Use the APISIX API Gateway to expose services to clients.
- Extend a provided Flutter client to add a new "cloud cash" feature that uses the services you have built.

You can access the sample CloudBank application at the following location: https://bit.ly/cloudbankAI.

Copyright © 1999, 2025, Oracle and/or its affiliates.

28

Developing Applications with Sagas

This chapter covers Saga APIs that allow applications that are built using microservices principles to incorporate efficient transactions across microservices. Saga APIs are implemented while closely following the Long Running Action (LRA) specification from Eclipse MicroProfile.

Topics:

- Implementing Sagas with Oracle Database
- Oracle Saga Framework Overview
- Saga Framework Features
- Saga Framework Concepts
- Initializing the Saga Framework
- Setting Up a Saga Topology
- Managing a Saga Using the PL/SQL Interface
- Developing Java Applications Using Saga Annotations
- Finalizing a Saga Explicitly

28.1 Implementing Sagas with Oracle Database

You can use Oracle Database as a platform to build your Saga-based, on-cloud or on-premises applications. When you implement Sagas with Oracle Database for handling distributed transactions, you can leverage the robust underlying infrastructure built into Oracle Database. The Saga infrastructure maintains the global application state for microservices, facilitating development and throughput.

The following are some key benefits of using Oracle Database for Saga implementation:

- The Saga implementation is integrated in the database to make Sagas simpler to code, deploy, and maintain.
- Simplified Saga annotations following the Eclipse LRA standard make it easy to enable transactions across microservices using Oracle Database.
- Data consistency is achievable across microservices.
- Auto-compensating logic based on lock-free, reservable columns is provided in the database to handle rollbacks on Saga rollback.
- Advanced Queuing (AQ) and Oracle Transactional Event Queues (TxEventQ) messaging platforms are built into Oracle Database, enabling microservices to produce and consume messages and events as part of database transactions.
- Support is provided for different languages, including PL/SQL and Java applications.
- The Saga framework recommends a JSON type for representing the application payload.
- There is improved availability and scalability for applications that require transactions across microservices. Such transactions are common with monolith applications that are

Copyright © 1999, 2025, Oracle and/or its affiliates.

converting to microservices, where transactions may span multiple bounded contexts. Note that a bounded context is an explicit boundary within which a business domain model exists.

28.2 Oracle Saga Framework Overview

The Oracle Saga framework provides the foundation to implement and administer Sagas for microservices applications built using Oracle Database. The framework provides administrative and client interfaces. Administrative interfaces enable Saga applications to configure and manage Saga participants and message brokers. Client interfaces enable applications to initiate, participate, and finalize Sagas. Saga participants are automatically configured with message channels and message propagation. The Saga framework is compensation-aware and provides for automatic rollback of the affected data when a Saga is rolled back. The Saga framework includes PL/SQL packages, dictionary tables, and Advanced Queuing (AQ) integration that facilitate Sagas in the database.

The Oracle Saga framework leverages two important features of Oracle Database. There is a transaction layer with reservable column support that enables recording of appropriate meta-information when reservable columns are updated. The reservable column support invokes compensating transactions automatically to revert the state of the reservable column updates when a Saga is rolled back. Reservable columns are a part of the lock-free reservation feature that Oracle provides to improve transactional concurrency.

Another important layer integrated into the database is the event queues built on the Oracle Advanced Queuing (AQ) technology. AQ provides the asynchronous messaging platform that connects various microservice participants. The Saga framework deploys a hub-and-spoke topology to connect Saga initiators, coordinators, and participants. The Saga message broker acts as the hub in this topology.

See Also

- Using Lock-Free Reservation
- *Oracle Database Advanced Queuing User's Guide*

28.3 Saga Framework Features

The Saga framework provides the following features for implementing Sagas on Oracle Database:

- PL/SQL administrative interfaces to add and manage active Sagas and Saga participants
- PL/SQL and Java interfaces for applications to interact with the Saga framework, and participate and finalize (commit or rollback) database Sagas
- Asynchronous message communication across multiple participants that contribute to a Saga, facilitated by microservice-specific AQ message propagation infrastructure from Oracle
- Reservable column-based Saga finalization (compensation and completion) support to enable recording of reservable column updates, and auto-compensation of transactions to roll back changes for canceled Sagas
- User-defined Saga finalization (compensation and completion) support to enable users to explicitly define and automatically execute Saga finalization

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Strong messaging semantics to ensure that interservice messages are delivered exactly once and never lost or duplicated
- Saga ID support to bind individual transactions to the Saga ID
- A globally unique name for each participant in a Saga
- Automatic configuration of message channels and message propagation for Saga participants that are provisioned into the system
- Various dictionary tables to maintain the status of Sagas
- System-defined views to report on Sagas and Saga participants in the system
- Eclipse Microprofile LRA specification emulation

28.4 Saga Framework Concepts

Saga *Participant* and *Initiator*

A Saga participant represents a spoke (participant microservice) in the Saga topology that either initiates and orchestrates a Saga or enrolls to participate in one. A participant is associated with a message broker (that is local or remote) and optionally with a local Saga coordinator. Saga participants that initiate Sagas are described in this document as "initiators" and non-initiator participants are described as "participants". Saga initiators must be associated with a Saga coordinator. In the initial version, the Saga framework only supports a local Saga coordinator (from the same PDB) for an initiator. Meanwhile, a Saga participant need not be associated with a coordinator and can be remote to the initiator.

Any participant can initiate a Saga. A Saga initiator is a microservice that initiates a Saga and enrolls other microservice participants by sending asynchronous messages.

In the travel agency example that follows, the participant microservices are the flight service, hotel service, and car service. A participant microservice executes one or more participant transactions on behalf of a Saga. Compensation action for a participant transaction is maintained in the local PDB. Participants are responsible for maintaining the local state of their Sagas.

Transaction Coordinator

A transaction coordinator acts as a transaction manager in the Saga topology. The Saga coordinator maintains the Saga state on behalf of the Saga initiator.

A Saga topology can have several Saga coordinators. In the initial release of the Saga framework, a Saga participant can only associate with coordinators who are local (same PDB and schema) to the participants. A coordinator can, however, associate with a local or remote message broker.

Message Broker

A message broker acts as an intermediary and represents the hub in the Saga topology. A broker provides a message delivery service for the message propagation between two or more Saga participants and their coordinator. Each Saga participant or coordinator is associated with a single broker, who can be local or remote. A broker does not maintain any Saga state.

Administrator and Client Interfaces

The Saga framework provides two PL/SQL packages: `DBMS_SAGA_ADM` and `DBMS_SAGA`.

The `DBMS_SAGA_ADM` package provides the administrative interface for the Saga framework. Using the administrative interface, you can manage Saga entities and ongoing Sagas.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The DBMS_SAGA package provides the developer APIs to initiate and complete Sagas, when building microservices applications.

Java developers, who want to create microservices using Sagas, should use Saga annotations, which is defined separately in this document.

Saga Annotations

Java applications initiating and participating in Sagas should use the Saga annotations for flexibility and simplicity of development. For more details, see Developing Java Applications Using Saga Annotations section of this document.

Advanced Queuing

The Saga Infrastructure leverages existing Oracle Advanced Queuing (AQ) technology, including message propagation between queues and event notification. AQ is the transaction communication mechanism for Saga participants and coordinators. AQ provides necessary plumbing and message channels to facilitate asynchronous message communication across multiple participants contributing to a Saga. AQ provides exactly-once message propagation without distributed transactions.

Reservable columns

A reservable column is a column type that has auto-compensating capabilities. Reservable columns enable individual databases to maintain reservation journals that record compensating actions for participant transactions. Compensating actions are automatically executed on the reservable column values in the event of a Saga rollback to revert local transactional changes.

Saga local transactions that the participant microservices execute can modify one or more reservable columns of database tables. Sagas leverage reservable columns to record compensating actions when transactional changes are made and automatically invoke the compensating actions when transactions fail. Compensating actions for reservable columns are recorded in reservation journals. Compensating transactions free up any resources that were locked earlier.

Dictionary Views

The Saga framework provides system-defined views to report on Sagas and Saga participants in the system. System defined views ALL_MICROSERVICES, CDB_MICROSERVICES, DBA_MICROSERVICES, and USER_MICROSERVICES provide the ability to monitor Sagas and Saga participants in the system.

The following views show all participants in the system:

- CDB_SAGA_PARTICIPANTS
- DBA_SAGA_PARTICIPANTS
- USER_SAGA_PARTICIPANTS

The following system-defined views show the dynamic state of ongoing Sagas:

- CDB_SAGAS
- DBA_SAGAS
- USER_SAGAS

The following system-defined views show the state of completed Sagas:

- CDB_HIST_SAGAS

Copyright © 1999, 2025, Oracle and/or its affiliates.

- DBA_HIST_SAGAS
- USER_HIST_SAGAS

Information for completed Sagas is retained for a period of 30 days. You can configure this duration using the saga_hist_retention database parameter.

The following system-defined views show incomplete Sagas:

- CDB_INCOMPLETE_SAGAS
- DBA_INCOMPLETE_SAGAS
- USER_INCOMPLETE_SAGAS

The following views provide details about reservable columns for incomplete Sagas:

- CDB_SAGA_PENDING
- DBA_SAGA_PENDING
- USER_SAGA_PENDING

The following views provide details about each Saga:

- CDB_SAGA_DETAILS
- DBA_SAGA_DETAILS
- USER_SAGA_DETAILS

The following views show the pending finalization actions for Sagas:

- CDB_SAGA_FINALIZATION
- DBA_SAGA_FINALIZATION
- USER_SAGA_FINALIZATION

> **See Also**
>
> Oracle Database Reference guide

Finalization Methods

Sagas are finalized when all participants commit (complete) or roll back (compensate) their respective participant transactions. Saga commit() or rollback() operations result in the completion of a Saga. The Saga framework enables you to also implement application-specific finalization in addition to the implicit reservable column-based finalization.

Eclipse Microprofile LRA Specification

The Saga framework emulates the Eclipse Microprofile LRA specification and provides equivalent functionality with certain limitations. The initial version of the Saga framework has the following limitations:

- Nested Sagas are not supported.
- Saga initiators and coordinators are co-located.

Copyright © 1999, 2025, Oracle and/or its affiliates.

28.5 Initializing the Saga Framework

Initializing Parameters

In the database initialization parameter file, `init.ora`, the `max_saga_duration` parameter identifies the time in seconds beyond which a Saga can be considered incomplete. The default value for this parameter is 86400 seconds. Any Saga that exceeds this configurable duration is considered incomplete and is liable to termination using the `dbms_saga.rollback_saga()` API. The Saga initiator automatically rolls back Sagas that exceed their duration.

In the `init.ora` file, you can use the `_use_saga_qtyp` parameter to set the message queuing type to be used in the Saga infrastructure. By default, the `_use_saga_qtyp` parameter is set to 0, which means it uses classic AQ queues, but if you want to switch to transactional event queues (TEQ), the `_use_saga_qtyp` parameter must be set to 1. Any change in the `_use_saga_qtyp` parameter must be done before creating the Saga entities: broker, coordinator, or participant. Changing the `_use_saga_qtyp` parameter after creating the Saga entities does not change the type of the queues used in the Saga.

> **See Also**
>
> *Oracle Database Transactional Event Queues and Advanced Queuing User's Guide*

Roles for Access Control

The Saga framework enables database administrators to provide the necessary privileges to the database users, who can administer and participate in Sagas. Users can have the following roles and privileges:

User Role	Access
`SAGA_ADM_ROLE`	Provides the ability to invoke APIs from the `DBMS_SAGA_ADM` package. This role is required for Saga administrators for the initial setup and provides full access to the `DBMS_SAGA_ADM` API. In the Saga framework setup example, `SAGA_ADM_ROLE` privilege is granted to the mailbox user MB.
`SAGA_PARTICIPANT_ROLE`	This role is required for Saga participant services. Saga primitives can only be invoked by a user that has the `SAGA_PARTICIPANT` role granted to it.
`SAGA_CONNECT_ROLE`	This role is provided to the remote dblink user.

`SYS` owns all the dictionary tables and user-accessible views of the data dictionary.

28.6 Setting Up a Saga Topology

The Saga framework provides administration APIs that the DBAs (with `SAGA_ADM_ROLE` privilege) can use to define and manage the Saga participants, coordinators, and brokers. The Saga PL/SQL package called `SYS.DBMS_SAGA_ADM` implements the Saga administrative interface. A participant must invoke the procedures in the `SYS.DBMS_SAGA_ADM` package from inside its schema and PDB.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The administrative interface provides the following APIs to provision Saga participants and brokers.

- `add_participant()` to add participants and their coordinator
- `add_broker()` to explicitly create a broker
- `add_coordinator()` to explicitly create a coordinator
- `drop_participant()` to drop a participant
- `drop_coordinator()` to drop a coordinator
- `drop_broker()` to drop a broker

Saga participants provisioned to the system are automatically configured with message channels and message propagation. Adding a participant creates a corresponding entry in the `SYS.SAGA_PARTICIPANT$` dictionary table, and creates incoming and outgoing Java topics for the participant. The inbound and outbound Java topics are provisioned with system-generated names that are derived from the participant's name. The following sections have further details about the administrative interface processing.

> **See Also**
>
> DBMS_SAGA_ADM for a complete description of `SYS.DBMS_SAGA_ADM` package APIs

28.6.1 Adding a Message Broker

In the Saga framework, a message broker acts as a message delivery service that receives messages from the Saga participants and propagates them to the Saga recipients.

The Saga framework uses the `dbms_saga_adm.add_broker()` API to add brokers explicitly to the framework. Creating a message broker creates a single Java topic that acts as the mailbox for the Saga participants. The system-generated name of the Java topic is of the form `SAGA$_<broker_name>_INOUT`, where the message broker name (`broker_name`) is the identifier that is provided as an input to the `add_broker()` call.

The `SAGA_ADM_ROLE` (administrator) role is required to create a Saga message broker.

28.6.2 Adding a Coordinator

A Saga coordinator acts as a transaction manager for Sagas. A Saga coordinator maintains the state of the Saga across various participants.

The Saga framework uses the `dbms_saga_adm.add_coordinator()` API to add coordinators to the framework.

> **Note**
>
> The `add_coordinator()` API must be called prior to invoking the `add_participant()` API, because `add_participant()` needs a coordinator name as an argument. Example: Saga Framework Setup explains the preparatory and provisioning steps.

The `add_coordinator()` interface executes the following:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Creates system-defined AQ queues for the coordinator's inbound and outbound message channels.
- Establishes a bidirectional message propagation channel between the coordinator and the message broker.

28.6.3 Adding a Participant

A participant is a named entity that represents an application or a microservice that desires to participate in database Sagas. Adding a participant sets up a bidirectional message propagation channel between the participant and the message broker.

The Saga framework uses the `dbms_saga_adm.add_participant()` API to add participants to the framework.

> **Note**
>
> Prior to invoking `add_participant()`, complete the pre-requisite steps in the participant and broker databases (PDBs). Example: Saga Framework Setup explains the preparatory and provisioning steps.

The `add_participant()` interface executes the following:

- Creates system-defined Java topics for the inbound and outbound message channels of the participants.
- Establishes a bidirectional message propagation channel between the participant and the message broker.

28.6.4 Managing Participants and Message Brokers

You can drop participants, coordinators, and message brokers using the `drop_participant()`, `drop_coordinator()`, and `drop_broker()` APIs. Dropping a participant, coordinator, or message broker also removes the associated JMS topic.

> **Note**
>
> - You can drop a participant only if there are no ongoing Sagas and no pending messages in the participant's incoming queue.
> - You can drop a coordinator only if there are no participants associated with the coordinator.
> - You can drop a message broker only if there are no registered participants and no pending messages.

> **See Also**
>
> DBMS_SAGA_ADM for details of administrative APIs in the `SYS.DBMS_SAGA_ADM` package

Copyright © 1999, 2025, Oracle and/or its affiliates.

28.6.5 Message Propagation

Message propagation transfers messages between Saga entities, namely initiators, participants, and brokers. Adding a participant to the Saga framework sets up message propagation. A message propagation job connects a participant's outbound topic to a broker's INOUT topic and connects a broker's INOUT topic to a participant's inbound topic.

- To propagate a participant's outbound topic to a broker's INOUT topic, the message propagation job uses the dblink from the `dblink_to_broker` parameter of the `add_participant()` interface.
- To propagate a broker's INOUT topic to a participant's inbound topic, the message propagation job uses the dblink from the `dblink_to_participant` of the `add_participant()` interface.

  Propagation of messages to a particular participant happens only for messages destined for that participant.

28.6.6 About Dictionary Tables

A globally unique identifier (GUID) is used to identify every Saga transaction. The `sys.saga_finalization$` dictionary table records individual steps that are required to complete the compensating actions for a Saga.

Several dictionary tables track the state (meta-data) associated with the Saga transactions at the participant database. These tables are:

- `sys.saga_message_broker$`: This table stores information for Saga brokers. Rows are inserted into the `saga_message_broker$` table using an explicit `add_broker()` call. Brokers can be remote or local, and this information is captured using the `saga_message_broker$.remote` column.
- `sys.saga_participant$`: This table stores information for participants and coordinators of the Saga framework. Rows are inserted into the `saga_participant$` table using the `add_participant()` or the `add_coordinator()` calls. Participants can be remote or local, and this information is captured using the `saga_participant$.remote` column.
- `sys.Saga$`: The `sys.Saga$` table contains entries for Sagas, either initiated on the given PDB or joined (using `joinSaga()`).
- `sys.saga_finalization$`: The `saga_finalization$` table at the participant database records the sequence number and reservation journal information for each unique reservable table updated as part of the participant transaction. The `saga_finalization$` table does not maintain information for application-specific compensation.
- `sys.saga_participant_set$`: A Saga initiator sends AQ JMS messages to enroll Saga participants. The Saga AQ JMS messages use special JMS message properties to indicate the `saga_id` and other Saga attributes to the participants. An entry in `sys.saga_participant_set$` table tracks each participant enrolled in a Saga. These entries track the enrollment and finalization status for each participant.
- `sys.saga_pending$`: This table records the reservation journal compensation information for Sagas that have timed out. The Saga infrastructure uses this information to forcefully commit or rollback a Saga.
- `sys.saga_errors$`: This table records error messages corresponding to various Sagas.

The following two database parameters affect Sagas:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `max_saga_duration`: This database parameter defines the maximum time in seconds that a Saga is considered to be active and is not marked incomplete. `max_saga_duration` is the system default for Saga duration, You can override this value using the `begin()` API. A Saga is marked incomplete if any of its participants are unable to finalize the Saga within the Saga duration. An incomplete Saga can be finalized using the `dbms_saga.rollback_saga()` interface.
- `saga_hist_retention`: This database parameter defines the maximum time in days for retaining information for completed Sagas. The default value for `saga_hist_retention` is 30 days.

28.6.7 Example: Saga Framework Setup

The following example configures a set of participant microservices and a broker.

This example sets up and configures a broker, two participants, namely TravelAgency and Airline, and their respective coordinator. Following these preparatory and provisioning steps, a Saga topology with two participants is created.

Preparatory Steps

Steps 1 through 7 are executed on `brokerPDB`, 8 through 11 at `TravelAgency`, and 12 through 15 at `AirlinePDB`.

1. Provision or designate a pluggable database (PDB) called `brokerPDB` to host the broker.
2. At `brokerPDB`, create a mailbox user that owns the broker and its JMS topic. For example: `MB`.
3. Grant the role `SAGA_ADM_ROLE` to user `MB`.
4. Create a proxy user `AirlineatMB` (Role: `SAGA_CONNECT_ROLE`).
5. Create a proxy user `TravelAgencyatMB` (Role: `SAGA_CONNECT_ROLE`).
6. Create dblink `LinkToAirline` to Airline PDB using `MBatAirline` schema.
7. Create dblink `LinkToTravelAgency` to Travel PDB using `MBatTravel` schema.
8. Provision or designate a PDB called `TravelAgency` to host Travel Reservation service.
9. At the `TravelAgency` PDB, create a user `TA`.
10. Create a proxy user `MBatTravelAgency` (Role: `SAGA_CONNECT_ROLE`).
11. Create dblink `LinktoBroker` to `brokerPDB` using `TravelAgencyatMB` schema.
12. Provision or designate a PDB called `AirlinePDB` to host the Airline.
13. At `AirlinePDB`, create user `Airline`.
14. Create proxy user `MBatAirline` (Role `SAGA_CONNECT_ROLE`).
15. Create dblink `LinktoBroker` to `brokerPDB` using `AirlineatMB` schema.

DBA views and entries associated with the dictionary entries in this example can be found in the `DBA_SAGA_PARTICIPANTS` view. For more information about the `DBA_SAGA_PARTICIPANTS` view and the following DBA views, see the Database Reference Guide.

- `DBA_SAGAS`
- `DBA_HIST_SAGAS`
- `DBA_INCOMPLETE_SAGAS`
- `DBA_SAGA_DETAILS`

Copyright © 1999, 2025, Oracle and/or its affiliates.

- DBA_PARTICIPANT_SET
- DBA_SAGA_FINALIZATION
- DBA_SAGA_PENDING
- DBA_SAGA_ERRORS

Provisioning Steps

BrokerPDB:

```
--Add a broker
dbms_saga_adm.add_broker(name=>'TravelBroker',  schema=>'MB');
```

TravelAgencyPDB:

```
--Add the Saga coordinator(local to the initiator)
 dbms_saga_adm.add_coordinator(
 coordinator_name => 'TACoordinator',
 dblink_to_broker => 'LinktoBroker',
 mailbox_schema => 'MB',
 broker_name => 'TravelBroker',
 dblink_to_coordinator => 'LinkToTravelAgency'
);

--Add the local Saga participant TravelAgency and its coordinator as below
 dbms_saga_adm.add_participant(
 participant_name=> 'TravelAgency',
 coordinator_name=> 'TACoordinator',
 dblink_to_broker=> 'LinktoBroker',
 mailbox_schema=> 'MB',
 broker_name=> 'TravelBroker',
 dblink_to_participant=> 'LinktoTravelAgency',
 callback_package => 'dbms_ta_cbk'
);
```

AirlinePDB:

```
--Add the local Saga participant Airline as below
 dbms_saga_adm.add_participant(
 participant_name=> 'Airline',
 dblink_to_broker=> 'LinktoBroker',
 mailbox_schema=> 'MB',
 broker_name=> 'TravelBroker',
 dblink_to_participant=> 'LinktoAirline'
 callback_package => 'dbms_airline_cbk'
);

--Add a table with reservable column which maintains flight information
Create table flights(id NUMBER primary key,
 seats NUMBER reservable constraint flights_const check (seats > 0));
```

28.7 Managing a Saga Using the PL/SQL Interface

Use PL/SQL to develop packaged microservice applications in the database without requiring a mid-tier.

The PL/SQL package DBMS_SAGA enables you to use PL/SQL to develop packaged microservice applications in the database without requiring a mid-tier to communicate with a

Copyright © 1999, 2025, Oracle and/or its affiliates.

database. The DBMS_SAGA package provides the PL/SQL interfaces that enable client programs to initiate and interact with database Sagas.

See Also

- DBMS_SAGA for the full description of the DBMS_SAGA package APIs
- Developing Java Applications Using Saga Annotations

28.7.1 Example: Saga PL/SQL Program

The following is a PL/SQL sample program depicting a Saga initiator (TravelAgency) and a Saga participant (Airline).

Example 28-1 Saga Initiator

```
declare
  saga_id RAW(16);
  request JSON;
begin
  saga_id := dbms_saga.begin_saga('TravelAgency');
  request := json('{"flight":"United"}');
  dbms_saga.send_request(saga_id, 'Airline', request);
end;
/
```

Example 28-2 Airline

```
create or replace package dbms_airline_cbk as
function request(saga_id in RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)
return JSON;
end dbms_airline_cbk;
/

create or replace package body dbms_airline_cbk as
function request(saga_id in RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)
return JSON as
response JSON;
tickets NUMBER;
BEGIN
  BEGIN
    select seats into tickets from flights where id = json_value(payload, '$.flight')
for update;
    IF tickets > 0 THEN
      response := json('{"result":"success"}');
      update flights set seats = seats - 1 where id = json_value(payload, '$.flight');
    ELSE
      response := json('{"result":"failure"}');
    END IF;
  EXCEPTION
      WHEN OTHERS THEN
        response := json('{"result":"failure"}');
  END;
    return response;
end;

end dbms_airline_cbk;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 28-3 TravelAgency

```
create or replace package dbms_ta_cbk as
procedure response(saga_id in RAW, saga_sender IN varchar2, payload IN JSON DEFAULT
NULL);
end dbms_ta_cbk;
/

create or replace package body dbms_ta_cbk as

procedure response(saga_id in RAW, saga_sender IN varchar2, payload IN JSON DEFAULT
NULL) as
booking_result VARCHAR2(10);
begin
  booking_result := json_value(payload, '$.result');
  IF booking_result = 'success' THEN
    dbms_saga.commit_saga('TRAVELAGENCY', saga_id);
  ELSE
    dbms_saga.rollback_saga('TRAVELAGENCY', saga_id);
  END IF;
end;
end dbms_ta_cbk;
/
```

28.8 Developing Java Applications Using Saga Annotations

Java applications initiating and participating in Sagas should use the Saga annotations for flexibility and simplicity of development. In this section, you can find further details on Saga and LRA annotations that Java applications can leverage.

> **Note**
>
> The Saga topology must be created separately.

> **See Also**
>
> Setting Up a Saga Topology for more information about creating a Saga topology

The following participant types are used with Saga annotations.

Saga Initiator

A Saga initiator is a special Saga participant that is responsible for the Saga lifecycle. The initiator initiates a Saga, invites other participants to join the Saga, and finalizes the Saga. Only the initiator can send messages requesting other participants to join a Saga. A Saga initiator is associated with a Saga coordinator. Saga initiators use the standard `@LRA` annotation defined by the MicroProfile LRA specification that allows JAX-RS applications to initiate Sagas. The Saga framework's support for `@LRA`, `@Complete`, and `@Compensate` annotations is similar to the corresponding LRA annotation in the MicroProfile LRA specification. Instead of an LRA ID, the Saga framework's `@LRA` annotation initializes and returns a Saga ID to the application. The Java class for initiator must extend the `SagaInitiator` class provided by this framework.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> The SagaInitiator class in the section titled "Saga Interfaces and Classes"

Saga Participant

A Saga participant joins a Saga at the request of a Saga initiator. A participant can directly communicate with the initiator but not with other participants. Saga participants use the `@Request` annotation to mark the method invoked for processing incoming Saga payloads. Such methods consume a `SagaMessageContext` object that contains all the required metadata and payload for the Saga. The `@Request` annotated method returns a JSON object that the Saga framework automatically sends to the Saga initiator in response. A participant class must extend `SagaParticipant`.

> **See Also**
>
> The SagaParticipant class and the SagaMessageContext class in the section titled "Saga Interfaces and Classes"

28.8.1 LRA and Saga Annotations

The Saga framework emulates the LRA framework and provides equivalent functionality. The following lists the annotations used in the Saga framework.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 28-1 LRA and Saga Annotations

Annotation	Description
`@LRA`	For Sagas, the LRA annotation controls the Saga initiation behavior. Currently, Sagas are handled using only an asynchronous model. The use of `end=true` is currently not supported. Generally, Saga finalization should be handled by calling `Saga.commitSaga()` or `Saga.rollbackSaga()` from a method that is annotated with `@Response`. The Saga implementation supports different LRA values, according to the LRA specification, except for annotation `@LRA.type.NESTED`. The Saga JAX-RS filter initializes a new Saga on behalf of the participant (described using the `@Participant` annotation), when needed. The `Saga ID` for the newly created Saga is inserted into the header parameter "Long-Running-Action" as defined by the specification. The method annotated with `@LRA` should be implemented by a class that extends the `SagaInitiator` class. This allows automatic instantiation of the Saga object as described by the `@LRA` annotation and the subsequent insertion of the LRA ID into the HTTP header. Participants can be invited to join a Saga using the `sendRequest()` method of the Saga class. For more information, see Saga Interface methods. Even if the initiator invokes `sendRequest()` to a participant multiple times for the same Saga, the participant joins the Saga only once. Currently, the Saga framework only supports `@LRA` on JAX-RS endpoints (using the JAX-RS filter) on the initiator. You can use the `beginSaga()` API to initiate Sagas manually.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 28-1 (Cont.) LRA and Saga Annotations

Annotation	Description

`@SagaConnection`	Any method annotated with `@SagaConnection` is used to fetch a JDBC Connection to create Saga Producers or Consumers while instantiating the Saga Participant.
	Each invocation of the method annotated with `@SagaConnection` should yield a new instance of a JDBC Connection.
	Note
	When `close()` is called on a participant, the connections borrowed by calling the method decorated with `@SagaConnection` are closed (a `connection.close()` is invoked). You can change this behavior by modifying the `closable()` field of `@SagaConnection`.
	`@SagaConnection(closable=true )`
	`public java.sql.Connection getConnection() { ..... }`

`@Complete`	`@Complete` is an LRA annotation that indicates the method the Saga framework automatically invokes when a Saga is committed. The Saga framework provides a `SagaMessageContext` object to the annotated method as an input argument. For more information, see the `SagaMessageContext` class. The method signature for a `@Complete` annotated method should be similar to: `public void airlineComplete(SagaMessageContext info)` . See the note at the end of this section for additional method signatures. The `@Complete` annotated method should finalize the local transactional changes on behalf of the Saga and clear any Saga state it was holding for possible Saga compensation. The method should use the connection object available in the `SagaMessageContext` class to perform any database actions and must not explicitly commit or roll back the database transaction.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 28-1 (Cont.) LRA and Saga Annotations

Annotation	Description
`@Compensate`	`@Compensate` is an LRA annotation that indicates the method the Saga framework automatically invokes when a Saga is rolled back. The Saga framework provides a `SagaMessageContext` object to the annotated method as an input argument. For more information, see the `SagaMessageContext` class. The method signature for a `@Compensate` annotated method should be similar to: `public void airlineCompensate(SagaMessageContext info)` See the note at the end of this section for additional method signatures. The `@Compensate` annotated method should compensate the local transactions performed during the Saga and clear any Saga state. The method should use the connection object available in the `SagaMessageContext` class to perform any database actions. The method should not explicitly commit or roll back the database transaction.
`@Participant`	`@Participant` is a Saga-specific annotation to indicate the method that maps a class to a Saga participant. The Saga participant must be previously defined within the database using the `dbms_saga_adm.add_participant()` API. The name of the participant also is used in the property file to associate additional parameters, such as the number of listeners and the number of publishers. **Note** If `@SagaConnection` uses variables that require instantiation of the class decorated with `@SagaParticipant`, the `delay()` field of `@SagaParticipant` should be set to `true`. This delays the creation of Saga Producers and Consumers until the `start()` method is called on the class decorated with `@SagaParticipant`.
`@Request`	`@Request` is a Saga-specific annotation to indicate the method that receives incoming requests from Saga initiators. The Saga framework provides a `SagaMessageContext` object as an input to the annotated method. If the participant is working with multiple initiators, an optional sender attribute can be specified (regular expressions are allowed) to differentiate between them.
`@Response`	`@Response` is a Saga-specific annotation to indicate the method that collects responses from Saga participants enrolled into a Saga using the `sendRequest()` API. The Saga framework provides a `SagaMessageContext` object as an input to the annotated method. If the initiator is working with multiple participants, an optional sender attribute can be specified (regular expressions are allowed) to differentiate between them.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 28-1 (Cont.) LRA and Saga Annotations

Annotation	Description
`@BeforeComplete`	`@BeforeComplete` is a Saga-specific annotation to indicate the method that is invoked during Saga finalization before a Saga is committed. The method annotated with `@BeforeComplete` is invoked before the automatic completion for any lock-free reservations performed by the Saga. The use of `@BeforeComplete` is optional.
`@BeforeCompensate`	`@BeforeCompensate` is a Saga-specific annotation to indicate the method that is invoked during Saga finalization before a Saga is rolled back. The method annotated with `@BeforeCompensate` is invoked before the automatic compensation for any lock-free reservations performed by the Saga. The use of `@BeforeCompensate` is optional.
`@InviteToJoin`	`@InviteToJoin` is a Saga-specific annotation that indicates the method that is invoked when the initiator requests that a particular participant can join a given Saga (using the `sendRequest()` API). If the method returns true, the participant joins the Saga. Otherwise, a negative acknowledgment is returned, and `@Reject` is invoked. The use of `@InviteToJoin` is optional.
`@Reject`	`@Reject` is a Saga-specific annotation to indicate the method that is invoked when: • A participant declines to join a Saga as defined by `@InviteToJoin`. • An unhandled exception is raised in the participant, in the method indicated by `@Request`. `@Reject` annotation is applicable only for an initiator. The method annotated with `@Reject` can perform bookkeeping for the Saga at the initiator level.

Note

For all annotations with an associated method, three method signatures are supported. For example, consider the following method signatures for the `@Request` annotation.

- `ProcessPayload (SagaMessageContext info)`
- `ProcessPayload (URI sagaId)`
- `ProcessPayload (URI sagaId, URI parentId)`

Since the Saga framework does not support `NESTED` Sagas, `parentId` is always null. For using these alternate signatures, a utility method: `getSagaMessageContext()` is provided to retrieve the `SagaMessageContext` object from a given Saga ID.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> - All database operations should be performed using the database connection object found in the `SagaMessageContext` class, and an explicit transaction commit or rollback is not supported.
> - If there are multiple matches on the sender value, an exception is raised.

28.8.2 Packaging

The Saga framework comprises two libraries:

- The Saga Client library
- The JAX-RS filter

The JAX-RS filter is only needed if the application is a JAX-RS application and wishes to initiate Sagas using the `@LRA` annotation.

The following are the relevant Maven coordinates for the two libraries:

Saga Client Library

```
<dependency>
  <groupId>com.oracle.database.saga</groupId>
  <artifactId>saga-core</artifactId>
  <version>${SAGA_VERSION}</version>
</dependency>
```

JAX-RS Filter

```
<dependency>
  <groupId>com.oracle.database.saga</groupId>
  <artifactId>saga-filter</artifactId>
  <version>${SAGA_VERSION}</version>
</dependency>
```

28.8.3 Configuration

JAX-RS Filter Configuration

The JAX-RS filter has the following parameters that can be configured using a property file (`sagafilter.properties`, `application.properties`, or both):

Property Name	Description
`osaga.filter.database.tnsAlias`	The TNS alias to use from the `tnsnames.ora` file
`osaga.filter.database.walletPath`	The path to the wallet that needs to be used for the filter connection
`osaga.filter.database.tnsPath`	The path to the location of the `tnsnames.ora` file

Copyright © 1999, 2025, Oracle and/or its affiliates.

Property Name	Description
`osaga.filter.initiator.publisherCount`	The total number of publishers to instantiate in the publisher pool that is used to initiate Sagas Each time a new Saga needs to be initiated, an existing publisher from the pool is used. Once the Saga is initiated, the publisher is released back into the pool.
`osaga.filter.initiator.name`	The `osaga.filter.initiator.name` property should match the `@Participant` annotation value as the filter acts on behalf of the initiator.

> **Note**
>
> The Saga JAX-RS filter should be the only JAX-RS LRA filter in the environment. Multiple LRA filters can cause inconsistency and unpredictable results.

Participants Configuration

Each of the participants has the following parameters that can be configured using a property file (`osaga.app.properties`, `application.properties`, or both):

Property Name	Description
`osaga.<participant_name>.tnsAlias`	The TNS alias to use from the `tnsnames.ora` file **Note** This parameter is not required if `@SagaConnection` is used.
`osaga.<participant_name>.walletPath`	The path to the wallet that needs to be used for the participant connection **Note** This parameter is not required if `@SagaConnection` is used.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Property Name	Description
`osaga.<participant_name>.tnsPath`	The path to the location of the `tnsnames.ora` file **Note** This parameter is not required if `@SagaConnection` is used.
`osaga.<participant_name>.numListeners`	The number of listeners assigned to this participant per queue partition. Listeners are used to process incoming messages. In practice, the number of listeners configured for a participant or an initiator depends on the number of other participants and initiators that can communicate with it. This number may have to be increased due to other factors, such as when the processing time for the payloads is higher or the number of concurrent requests increase.
`osaga.<participant_name>.numPublishers`	Publishers are needed to send the requests to participants. In practice, the number of publishers depends on the expected number of concurrent Sagas. The `<participant_name>` is derived from the `@Participant` annotation. For example, a class marked with `@Participant(name="TravelAgency")` is configured using the `osaga.travelagency` prefix, for example, `osaga.travelagency.numListeners=2`.

Note

`numListeners` and `numPublishers` are independent entities. `numListeners` refers to the number of threads responding to incoming requests. `numPublishers` refers to an initiator's publisher threads. For an initiator, it is ideal to have `numListeners=numPublishers` for reliable throughput.

28.8.4 Saga Interface and Classes

The Saga interface and classes describe the functionality required to support Sagas. This section describes the interface and classes for Java applications.

28.8.4.1 Saga Interface

The Saga interface provides the means to participate in, complete, and request metadata for a database Saga. The Saga interface provides the following methods:

Copyright © 1999, 2025, Oracle and/or its affiliates.

`sendRequest(String recipient, String payload)` Method

Syntax:

```
* @param recipient - name of the participant to be enrolled
* @param payload - saga payload
public void sendRequest(String recipient, String payload)
```

Description: The `sendRequest(String recipient,String payload)` method is invoked at the initiator level to send a message (payload) to a participant. If the participant joins the Saga, the initiator invokes the method annotated with `@Response`. Otherwise, the initiator invokes the method annotated with `@Reject`.

> **Note**
>
> The `sendRequest(String recipient,String payload)` method automatically commits the transaction by default when used outside the scope of Saga-annotated methods. Fine-grained control over the transaction boundary can be achieved using the `sendRequest(java.sql.Connection connection, String recipient, String payload)` method.

`sendRequest(AQjmsSession session, TopicPublisher publisher, String recipient, String payload)` Method

> **Note**
>
> This method is deprecated. Instead, use the `sendRequest(java.sql.Connection connection, String recipient, String payload)` method.

Syntax:

```
* @param session - user supplied saga session
* @param publisher - user supplied topic publisher
* @param recipient - name of the participant to be enrolled
* @param payload - saga payload
public void sendRequest(AQjmsSession session, TopicPublisher publisher,
String recipient, String payload)
```

Description: This method is identical to `sendRequest(String recipient,String payload)` except for the use of `AQjmsSession` and `TopicPublisher` parameters.

Multiple `sendRequest()` calls can be made in a single database transaction, provided they have identical `AQjmsSession` session and `TopicPublisher` publisher parameter values. Other database operations (such as DML) can also be part of the same transaction, provided they use the database connection embedded in `AQjmsSession` session obtained using the `getDBConnection()` method of the `AQjmsSession` class. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `AQjmsSession`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> A `TopicPublisher` (publisher) instance is obtained using the `getSagaOutTopicPublisher` (`AQjmsSession` session) method declared in the `SagaInitiator` class.

`sendRequest(java.sql.Connection connection, String publisher, String recipient, String payload)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param publisher - user supplied topic publisher
* @param recipient - name of the participant to be enrolled
* @param payload - saga payload
public void sendRequest(java.sql.Connection connection, String publisher,
String recipient, String payload)
```

Description: This method is identical to `sendRequest(String recipient,String payload)` except for the use of `java.sql.Connection connection` and the `publisher` parameter.

Multiple `sendRequest()` calls can be made in a single database transaction, provided they use the supplied `java.sql.Connection` connection. Other database operations (such as DML) can also be part of the same transaction, provided they use the supplied database connection. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `Connection`.

`getSagaId()` Method

Syntax:

```
public String getSagaId()
```

Description: The `getSagaId()` method returns the Saga identifier associated with the Saga object instantiated using `beginSaga()` or `getSaga(String sagaId)` method in the `SagaInitiator` class.

`commitSaga()` Method

Syntax:

```
public void commitSaga()
```

Description: The `commitSaga()` method is invoked by the initiator to commit the Saga. As part of its execution, the methods annotated with `@BeforeComplete` and `@Complete` are invoked at both the initiator and participants levels. Reservable column operations, if any, are finalized between the `@BeforeComplete` and `@Complete` calls.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> The `commitSaga()` method auto commits the transaction by default, when used outside the scope of Saga-annotated methods. Fine-grained control over the transaction boundary can be achieved using the `commitSaga(java.sql.Connection connection)` method.

`commitSaga(AQjmsSession session)` Method

> **Note**
>
> This method is deprecated. Instead, use the `commitSaga(java.sql.Connection connection)` method.

Syntax:

```
* @param session - user supplied saga session
public void commitSaga(AQjmsSession session)
```

Description: This method is identical to the `commitSaga()` method except for the use of `AQjmsSession` session.

Other database operations (such as DML) can also be part of the same transaction, provided they use the database connection embedded in the `AQjmsSession` session obtained using the `getDBConnection()` method of the `AQjmsSession` class. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `AQjmsSession`.

`commitSaga(java.sql.Connection connection)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
public void commitSaga(java.sql.Connection connection)
```

Description: This method is identical to the `commitSaga()` method except for the use of `java.sql.Connection` connection.

Other database operations (such as DML) can also be a part of the same transaction, provided they use the supplied database connection. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `Connection`.

`commitSaga(boolean force)` Method

Syntax:

```
* @param force - force commit flag
public void commitSaga(boolean force)
```

Description: This method is identical to `commitSaga()` except for the use of the `force` flag.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The `force` flag could be used by a Saga participant in special situations to locally commit the Saga and inform the Saga coordinator without waiting for the finalization from the Saga initiator.

`commitSaga(AQjmsSession session, boolean force)` Method

> **Note**
>
> This method is deprecated. Instead, use the `commitSaga(java.sql.Connection connection, boolean force)` method.

Syntax:

```
* @param session - user supplied saga session
* @param force - force commit flag
public void commitSaga(AQjmsSession session, boolean force)
```

Description: This method is identical to the `commitSaga()` method except for the use of the `AQjmsSession` session and the `force` flag. It combines the functionality of the `commitSaga(AQjmsSession session)` and `commitSaga(boolean force)` methods.

> **Note**
>
> The `commitSaga(AQjmsSession session)`, `commitSaga(boolean force)`, and `commitSaga(AQjmsSession session, boolean force)` methods can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after the execution of the method.

`commitSaga(java.sql.Connection connection, boolean force)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param force - force commit flag
public void commitSaga(java.sql.Connection connection, boolean force)
```

Description: This method is identical to the `commitSaga()` method except for the use of the `java.sql.Connection` connection and the `force` flag. It combines the functionality of the `commitSaga(java.sql.Connection connection)` and `commitSaga(boolean force)` methods.

> **Note**
>
> The `commitSaga(java.sql.Connection connection)`, `commitSaga(boolean force)`, and `commitSaga(java.sql.Connection connection, boolean force)` methods can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after the execution of the method.

Copyright © 1999, 2025, Oracle and/or its affiliates.

`rollbackSaga()` Method

Syntax:

```
public void rollbackSaga()
```

Description: The `rollbackSaga()` method rolls back the Saga and invokes the methods annotated with `@BeforeCompensate` and `@Compensate` annotations. Reservable column operations (if any) are finalized between the `@BeforeCompensate` and `@Compensate` calls.

> **Note**
>
> The `rollbackSaga()` method auto commits the transaction by default if used outside the scope of Saga-annotated methods. Fine-grained control over the transaction boundary can be achieved using the `rollbackSaga(java.sql.Connection connection)` method.

`rollbackSaga(AQjmsSession session)` Method

> **Note**
>
> This method is deprecated. Instead, use the `rollbackSaga(java.sql.Connection connection)` method.

Syntax:

```
* @param session - user supplied saga session
public void rollbackSaga(AQjmsSession session)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `AQjmsSession` session.

Other database operations (such as DML) can also be part of the same transaction provided they use the database connection embedded in the `AQjmsSession` session obtained using the `getDBConnection()` method of the `AQjmsSession` class. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `AQjmsSession`.

> **Note**
>
> The `rollbackSaga(AQjmsSession session)` method can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction by default which is committed or rolled back after execution of the method.

Copyright © 1999, 2025, Oracle and/or its affiliates.

`rollbackSaga(java.sql.Connection connection)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
public void rollbackSaga(java.sql.Connection connection)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `java.sql.Connection` connection.

Other database operations (such as DML) can also be part of the same transaction provided they use the supplied database connection. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `Connection`.

> **Note**
>
> The `rollbackSaga(java.sql.Connection connection)` method can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction by default which is committed or rolled back after execution of the method.

`rollbackSaga(boolean force)` Method

Syntax:

```
* @param force - force rollback flag
public void rollbackSaga(boolean force)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `force` flag.

The `force` flag could be used by a Saga participant in special situations to locally roll back the Saga and inform the Saga coordinator without waiting for the finalization from the Saga initiator.

`rollbackSaga(AQjmsSession session, boolean force)` Method

> **Note**
>
> This method is deprecated. Instead, use the `rollbackSaga(java.sql.Connection connection, boolean force)` method.

Syntax:

```
* @param session - user supplied saga session
* @param force - force commit flag
public void rollbackSaga(AQjmsSession session, boolean force)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Description: This method is identical to the `rollbackSaga()` method except for the use of the `AQjmsSession` session and the `force` flag. It combines the functionality of the `rollbackSaga(AQjmsSession session)` and `rollbackSaga(boolean force)` methods.

> **Note**
>
> The `rollbackSaga(AQjmsSession session)`, `rollbackSaga(boolean force)`, and `rollbackSaga(AQjmsSession session, boolean force)` can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after execution of the method.

`rollbackSaga(java.sql.Connection connection, boolean force)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param force - force commit flag
public void rollbackSaga(java.sql.Connection connection, boolean force)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `java.sql.Connection` connection and the `force` flag. It combines the functionality of the `rollbackSaga(java.sql.Connection connection)` and `rollbackSaga(boolean force)` methods.

> **Note**
>
> The `rollbackSaga(java.sql.Connection connection)`, `rollbackSaga(boolean force)`, and `rollbackSaga(java.sql.Connection connection, boolean force)` can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after execution of the method.

`isSagaFinalized()` Method

Syntax:

```
public boolean isSagaFinalized()
```

Description: The `isSagaFinalized()` method can be invoked at an initiator or participant level and it checks whether the Saga has reached one of the finalization states. It returns false if the saga is in a `JOINING`, `JOINED`, or `TIMEDOUT` state, and returns true, otherwise.

`beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` Method

> **Note**
>
> This method is deprecated. Instead, use the `beginSagaTransaction(java.sql.Connection connection)` method.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax:

```
* @param session - user supplied saga session
* @param publisher - user supplied topic publisher
public void beginSagaTransaction(AQjmsSession session, TopicPublisher
publisher)
```

Description: The `beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` method is used outside the scope of a Saga-annotated method (for example, under the `@LRA annotated` method) to start a new Saga transaction. It can only be invoked at an initiator level. All operations performed inside the Saga transaction are committed or rolled back explicitly by calling the `commit()` or `rollback()` method of `AQjmsSession`. The `beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` call requires a corresponding `endSagaTransaction()` call to complete the Saga transaction.

> **Note**
>
> The `beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` method is serialized with the invocation of other Saga related methods, such as `commitSaga()` or `rollbackSaga()`.

`beginSagaTransaction(java.sql.Connection connection, String publisher)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param publisher - user supplied topic publisher
public void beginSagaTransaction(java.sql.Connection connection, String
publisher)
```

Description: The `beginSagaTransaction(java.sql.Connection connection)` method is used outside the scope of a Saga-annotated method (for example, under the `@LRA annotated` method) to start a new Saga transaction. It can only be invoked at an initiator level. All operations performed inside the Saga transaction are committed or rolled back explicitly by calling the `commit()` or `rollback()` method in `Connection`. The `beginSagaTransaction(java.sql.Connection connection)` call requires a corresponding `endSagaTransaction()` call to complete the Saga transaction.

> **Note**
>
> The `beginSagaTransaction(java.sql.Connection connection)` method is serialized with the invocation of other Saga related methods, such as `commitSaga()` or `rollbackSaga()`.

`endSagaTransaction()` Method

Syntax:

```
public void endSagaTransaction()
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Description: The `endSagaTransaction()` method ends the Saga transaction started by a `beginSagaTransaction()` call. The `endSagaTransaction()` method releases the lock upon a Saga if the Saga transaction was started with the lock flag set to true.

28.8.4.2 SagaMessageContext Class

The `SagaMessageContext` object is passed as an argument to methods annotated with Saga annotations upon a Saga related event. The `SagaMessageContext` object can be used to fetch metadata associated with the Saga that triggered the Saga annotated method.

The `SagaMessageContext` class provides the following methods.

`getSagaId()` Method

Syntax: `public String getSagaId()`

Description: The `getSagaId()` method returns the identifier of the Saga.

`getSender()` Method

Syntax: `public String getSender()`

Description: The `getSender()` method returns the name of the sender of the message.

`getPayload()` Method

Syntax: `public String getPayload()`

Description: The `getPayload()` method returns the payload associated with the Saga message.

`getConnection()` Method

Syntax: `public java.sql.Connection getConnection()`

Description: The `getConnection()` method can be used in any Saga-annotated method to return the database connection object of the Saga message listener thread. It can be used at an initiator or participant level. The connection can be used by the application to perform database operations.

Explicit `commit()` or `rollback()` in the saga-annotated method is not supported.

`getSaga()` Method

Syntax: `public Saga getSaga()`

Description: The `getSaga()` method returns the Saga object associated with the Saga that triggered the Saga annotated method. The Saga object can be used to finalize that Saga or request metadata for that Saga.

28.8.4.3 SagaParticipant Class

The `SagaParticipant` class provides the means to create and manage a Saga participant instance. The Saga participant is created using the `add_participant()` PL/SQL API provided in the `DBMS_SAGA_ADM` package.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> DBMS_SAGA_ADM for a complete description of the `SYS.DBMS_SAGA_ADM` package APIs.

The `SagaParticipant` class exposes the following methods.

`addSagaMessageListener()` Method

Syntax:

```
public void addSagaMessageListener()
```

Description: The `addSagaMessageListener()` method allows a Saga participant to add an additional message listener thread. Adding more Saga message listener threads allows the application to increase concurrency when processing Saga messages.

> **Note**
>
> `addSagaMessageListener()` adds only one message listener thread. To add more than one listener thread at once, the `addSagaMessageListener(int numListeners)` method should be used.

`addSagaMessageListener(int numListeners)` Method

Syntax:

```
* @param numListeners - number of listeners to add
public void addSagaMessageListener(int numListeners)
```

Description: The `addSagaMessageListener(int numListeners)` method allows a Saga participant to add additional message listener threads as `numListeners`. Addition of more Saga message listener threads allows for processing more Saga messages concurrently.

`removeSagaMessageListener()` Method

Syntax:

```
public void removeSagaMessageListener()
```

Description: The `removeSagaMessageListener()` method allows a Saga participant to remove a message listener thread. Under lower load, Saga message listener threads can be reduced to free up system resources.

> **Note**
>
> The `removeSagaMessageListener()` method removes only one message listener thread. To remove more than one listener thread at once the `removeSagaMessageListener (int numListeners)` method should be used.

Copyright © 1999, 2025, Oracle and/or its affiliates.

`removeSagaMessageListener(int numListeners)` Method

Syntax:

```
* @param numListeners - number of listeners to remove
public void removeSagaMessageListener(int numListeners)
```

Description: The `removeSagaMessageListener(int numListeners)` method allows a Saga participant to remove `numListeners` message listener threads. Under lower load, Saga message listener threads can be reduced to free up system resources.

`removeAllSagaMessageListeners()` Method

Syntax:

```
public void removeAllSagaMessageListeners()
```

Description: The `removeAllSagaMessageListeners()` method allows a Saga participant to remove all Saga message listener threads. Invoking this method prevents the execution of the methods annotated with Saga-specific annotations upon receiving a Saga message. This method can be used where the business logic of the methods annotated with Saga annotations needs to be changed. Any unprocessed Saga messages are retained in the Saga message queues and can be processed once message listeners restart.

`getSagaMessageListenerCount()` Method

Syntax:

```
public int getSagaMessageListenerCount()
```

Description: The `getSagaMessageListenerCount()` method returns the total number of Saga message listener threads associated with a Saga participant.

> **Note**
>
> In case of an AQ queue setup, `getSagaMessageListenerCount()` returns the message listener threads associated with a single partition of the Saga participant. Total message listener threads are obtained by `getSagaMessageListenerCount() * numPartitions`, which are defined during the `add_participant` call. The number of partitions configured for a participant can also be queried using the `DBA_SAGA_PARTITIPANTS` dictionary view.

`getSaga(String sagaId)` Method

Syntax:

```
* @param sagaId - Identifier for the saga to be retrieved
public Saga getSaga(String sagaId)
```

Description: The `getSaga(String sagaId)` method returns the Saga object associated with the Saga with the specified Saga identifier.

Copyright © 1999, 2025, Oracle and/or its affiliates.

`close()` Method

Syntax:

```
public void close()
```

Description: The `close()` method removes all message listener threads and connections associated with a Saga participant. When the Saga participant is closed, methods annotated with Saga annotations are not invoked any further. The Saga participant cannot participant in, complete, and request metadata for a database Saga. The pending Saga messages for the participant are retained and can be consumed in the future.

28.8.4.4 SagaInitiator Class

The `SagaInitiator` class provides the means to create and manage a Saga initiator instance. The Saga initiator is created using the `add_participant()` PL/SQL API provided in the `dbms_saga_adm` package. The `SagaInitiator` class inherits all methods specified in the `SagaParticipant` class. Additionally, the `SagaInititor` class exposes the following methods.

`beginSaga()` Method

Syntax:

```
public Saga beginSaga()
```

Description: The `beginSaga()` method starts a Saga and returns a `sagaId`. The `sagaId` can be used to enroll other participants.

> **Note**
>
> The `beginSaga()` method starts a Saga with the default timeout of 84600 seconds. Fine-grained control over the timeout can be achieved using `beginSaga(int timeout)`.

`beginSaga(int timeout)` Method

Syntax:

```
* @param timeout - saga timeout
public Saga beginSaga(int timeout)
```

Description: The `beginSaga(int timeout)` method starts a Saga and returns a Saga identifier with a user-specified timeout. The Saga identifier can be used to enroll other participants. Note: If the Saga is not finalized before the specified timeout, the Saga automatically finalizes depending on the database initialization parameter `_saga_timeout_operation_type(default=rollback)`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

`addSagaMessagePublishers(int numPublishers)` Method

Syntax:

```
* @param numPublisher - number of publishers to add
public void addSagaMessagePublishers(int numPublishers)
```

Description: The `addSagaMessagePublishers(int numPublishers)` method allows a Saga initiator to add additional message publisher threads as `numPublisher`. Addition of more Saga message publisher threads allows the initiator to create and manage more Sagas concurrently.

`removeSagaMessagePublishers(int numPublishers)` Method

Syntax:

```
* @param numPublisher - number of publishers to remove
public void removeSagaMessagePublishers(int numPublishers)
```

Description: The `removeSagaMessagePublishers(int numPublishers)` method allows a Saga initiator to remove `numPublishers` message publisher threads. Under lower load, Saga message publisher threads can be reduced to free up system resources.

`removeAllSagaMessagePublishers()` Method

Syntax:

```
public void removeAllSagaMessagePublishers()
```

Description: The `removeAllSagaMessagePublishers()` method allows a Saga initiator to remove all Saga message publisher threads. All message publisher threads can be removed if the initiator does not want to initiate or manage any Sagas further. Removing Saga message publishers does not affect the pending messages previously published.

`getSagaMessagePublisherCount()` Method

Syntax:

```
public int getSagaMessagePublisherCount()
```

Description: The `getSagaMessagePublisherCount()` method return the total number of saga message publisher threads associated with a saga initiator.

> **Note**
>
> In case of a AQ queue setup, the `getSagaMessagePublisherCount()` method returns the message publisher threads associated with a single partition of the Saga participant. Total message publisher threads are obtained by `getSagaMessagePublisherCount() * numPartitions`, which are defined during the `add_participant()` call. The number of partitions configured for a participant can also be queried using the `DBA_SAGA_PARTITIPANTS` dictionary view.

Copyright © 1999, 2025, Oracle and/or its affiliates.

`getSagaOutTopicPublisher(AQjmsSession session, int partition)` Method

Syntax:

```
* @param session - user supplied saga session
* @param partition - partition for which the publisher is needed
public TopicPublisher getSagaOutTopicPublisher(AQjmsSession session, int
partition)
```

Description: The `getSagaOutTopicPublisher(AQjmsSession session, int partition)` method returns the Saga topic publisher instance. The `TopicPublisher` instance can be supplied as a parameter to various methods that accept a user defined `AQjmsSession` to manage the transaction boundary manually.

> **Note**
>
> For AQ queue setup, the value of partition could be `[1-numPartitions]` specified in the `add_participant` call. In case of a TxEventQ queue setup, the value of partition should always be 1.

28.8.5 Example Program

The following example illustrates the use of a subset of the important Saga annotations through a simple Travel Agency Saga application. The Travel Agency emulates a real travel agency that performs Airline reservations for its end users. The Travel Agency is a JAX-RS application (Initiator) that uses one participant: Airline. Airline is implemented as a standard Java application (not JAX-RS).

The Travel Agency and Airline leverage appropriate Saga annotations for performing their tasks. In this example, `TravelAgency` is the Saga initiator that initiates a Saga to purchase airline tickets for its end users. `Airline` is the Saga participant. The example is a simple illustration where only one reservation is made by a participant towards a Saga. In practice, a Saga may span multiple participants.

> **Note**
>
> An application developer only needs to implement the annotated methods as shown in this example. The Saga framework provides the necessary communication and maintains the state of the Saga across a distributed topology.

> **Note**
>
> The `@LRA`, `@Compensate`, `@Complete` annotations are LRA annotations, whereas `@participant`, `@Request`, and `@Response` are Saga annotations.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 28-4 TravelAgency (Saga Initiator)

```
@Participant(name = "TravelAgency")
/* @Participant declares the participant's name to the saga framework */
public class TravelAgencyController extends SagaInitiator {
/* TravelAgencyController extends the SagaInitiator class */
 @LRA(end = false)
 /* @LRA annotates the method that begins a saga and invites participants */
 @POST("booking")
 @Consumes(MediaType.TEXT_PLAIN)
 @Produces(MediaType.APPLICATION_JSON)
  public jakarta.ws.rs.core.Response booking(
      @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
      String bookingPayload) {
    Saga saga = this.getSaga(lraId.toString());
    /* The application can access the sagaId via the HTTP header
       and instantiate the Saga object using it */
    try {
      /* The TravelAgency sends a request to the Airline sending
         a JSON payload using the Saga.sendRequest() method */
      saga.sendRequest ("Airline", bookingPayload);
      response = Response.status(Response.Status.ACCEPTED).build();
    } catch (SagaException e) {
      response=Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
    }
  }
  @Response(sender = "Airline.*")
  /* @Response annotates the method to receive responses from a specific
     Saga participant */
  public void responseFromAirline(SagaMessageContext info) {
    if (info.getPayload().equals("success")) {
      saga.commitSaga ();
      /* The TravelAgency commits the saga if a successful response is
received */
    } else {
      /* Otherwise, the TravelAgency performs a Saga rollback  */
      saga.rollbackSaga ();
    }
  }
}
```

Example 28-5 Airline (Saga Participant)

```
@Participant(name = "Airline")
/* @Participant declares the participant's name to the saga framework */
public class Airline extends SagaParticipant {
/* Airline extends the SagaParticipant class */
  @Request(sender = "TravelAgency")
  /* The @Request annotates the method that handles incoming request from a
given
     sender, in this example the TravelAgency */
  public String handleTravelAgencyRequest(SagaMessageContext
      info) {
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    /* Perform all DML with this connection to ensure
       everything is in a single transaction */
    FlightService fs = new
      FlightService(info.getConnection());
    fs.bookFlight(info.getPayload(), info.getSagaId());
    return response;
    /* Local commit is automatically performed by the saga framework.
       The response is returned to the initiator */
  }

  @Compensate
  /* @Compensate annotates the method automatically called to roll back a
saga */
  public void compensate(SagaMessageContext info) {
    fs.deleteBooking(info.getPayload(),
        info.getSagaId());
  }
  @Complete
  /* @Complete annotates the method automatically called to commit a saga */
  public void complete(SagaMessageContext info) {
    fs.sendConfirmation(info.getSagaId());
  }
}
```

> **Note**
>
> The import directives are not included in the sample program.

Other microservices, such as `Hotel` and `CarRental` would have a structure similar to the Airline microservice.

The Travel Agency could be modified in the following ways to accommodate another participant. Consider adding a Car Rental service:

- In TravelAgencyController's booking method, another `sendRequest()` is required to send the payload to the "Car" participant.

  ```
  Saga.sendRequest ("Car", bookingPayload.getCar().toString());
  ```

- A new annotated method is required to handle responses from the Car service.

  ```
  @Response(sender = "Car”)
  ```

- Additional logic in the `@Response` methods is necessary to account for the state of the reservation after messages are received from participants.

  ```
  @Response(sender = "Car")
  public void responseFromCar(SagaMessageContext info) {
    if (!info.getPayload().equals("success")) {
      /* On error, rollback right away */
      Saga.rollbackSaga ();
    } else {
      /* Store the car's response */
  ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

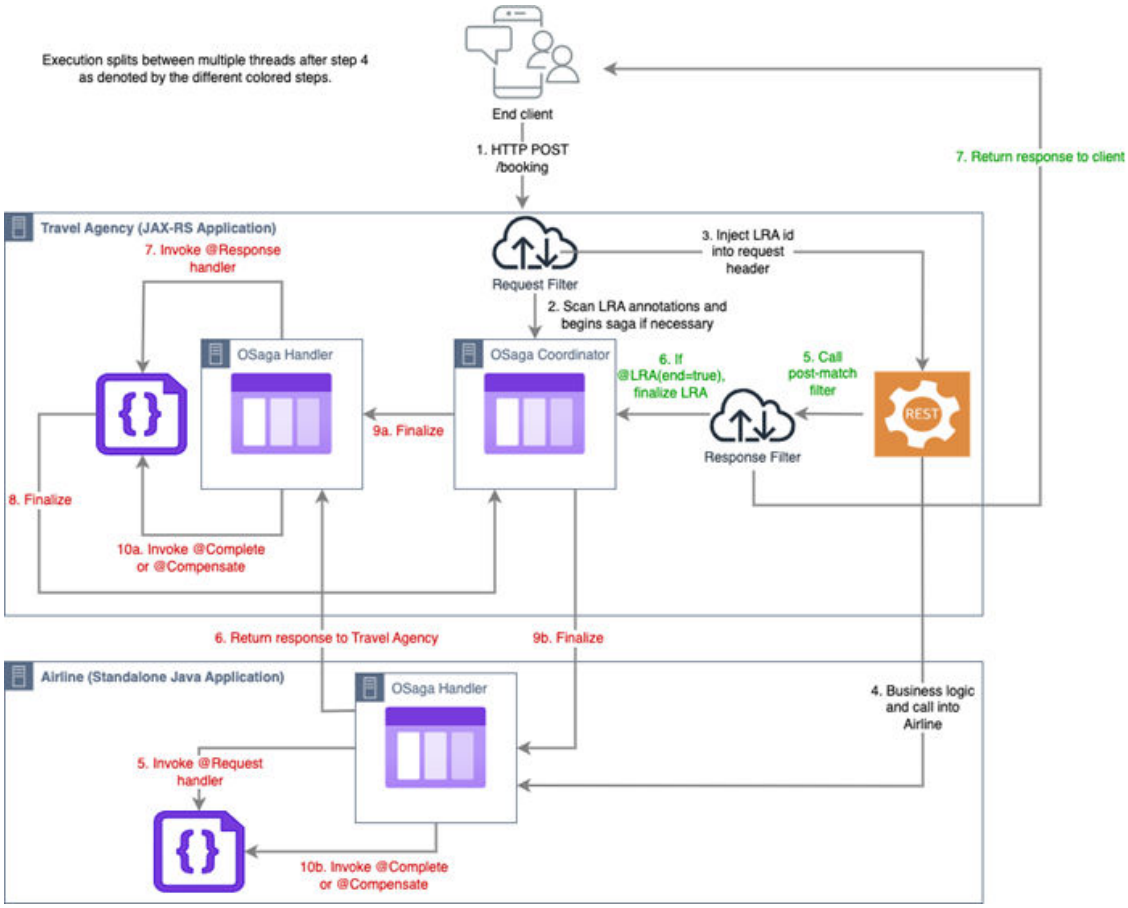
```
        cache.putCarResponse(info.getSagaId(), info.getPayload());

        /* Check to see if Airline has responded
           If a response is found, commit */
        if (cache.containsAirlineResponse(info.getSagaId())) {
          Saga.commitSaga ();
        }
      }
    }
```

The method to handle Airline's response would be similar, except that it would replace `cache.containsAirlineResponse(info.getSagaId())` with `cache.containsCarResponse(info.getSagaId())` and `cache.putAirlineResponse()` with `cache.putCarResponse()`.

The following asynchronous flow diagram shows the process flow of a Saga application illustrated in the previous code examples. The numbers in the flow diagram correspond to the numbers shown in the code snippet.

> **See Also**
>
> Appendix: Troubleshooting the Saga Framework for more information about the Saga lifecycle

Figure 28-1 Asynchronous Flow



Copyright © 1999, 2025, Oracle and/or its affiliates.

Saga Property File

The following shows the Saga property file used by the example program.

```
# JAX-RS filter properties
osaga.filter.database.tnsAlias=ta
osaga.filter.database.walletPath=/Path/to/wallet
osaga.filter.database.tnsPath=/Path/to/tns
osaga.filter.initiator.publisherCount=2
osaga.filter.initiator.name=TravelAgency
osaga.travelagency.tnsAlias=ta

# Property values for the Initiator Travelagency
osaga.travelagency.walletPath=/Path/to/wallet
osaga.travelagency.tnsPath=/Path/to/tns
osaga.travelagency.numListeners=2
osaga.travelagency.numPublishers=2

# Properties pertaining to the Airline participant
osaga.airline.tnsAlias=airline1
osaga.airline.walletPath=/Path/to/wallet
osaga.airline.tnsPath=/Path/to/tns
osaga.airline.numListeners=2
osaga.airline.numPublishers=2
```

28.9 Finalizing a Saga Explicitly

In addition to the automatic reservable column-based finalization, the Saga framework supports finalization (complete or compensate) that is application-specific and explicit (user-defined). The framework automatically invokes application-specific finalization routines during Saga finalization.

You can designate the application-specific finalization using the PL/SQL callback routines for the PL/SQL clients.

> **Note**
>
> Ensure that there is no explicit `COMMIT` command issued from the user-defined callbacks that are implemented using the PL/SQL callback package. The Saga framework commits the changes on behalf of the application.

For Java programs, Developing Java Applications Using Saga Annotations describes the `@BeforeComplete`, `@BeforeCompensate`, and `@Complete`, `@Compensate` annotations that a Java program can use to implement application-specific compensation. In such cases, the lock-free reservations are automatically compensated between the `@BeforeCompensate` and `@Compensate` annotated methods.

> **See Also**
>
> - Using Lock-Free Reservation
> - Integration with Lock-Free Reservation

Copyright © 1999, 2025, Oracle and/or its affiliates.

28.9.1 PL/SQL Callbacks for a PL/SQL Client

A PL/SQL based initiator or participant needs to implement a callback package to participate in Sagas. The callback package is called internally from the notification callback for the respective participant. The callback package enables you to ignore the internal details of the Saga infrastructure and focus on your business logic.

Applications must ensure that the implementation of a callback package does not invoke any database transaction control operations (commit/rollback).

The callback package provides the following functions:

`request` Function

Syntax: `function request(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL) return JSON;`

Description: The request function is invoked when receiving a Saga message with opcode `REQUEST`. The application is expected to implement this method. A JSON payload is returned that is sent back to the initiator as a response.

`response` Procedure

Syntax: `procedure response(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The response procedure is invoked when receiving a Saga message with opcode `RESPONSE`. The application is expected to implement this method.

`before_commit` Procedure

Syntax: `procedure before_commit(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `before_commit` procedure is invoked before executing the Saga commit on the current participant when receiving a Saga message with opcode `COMMIT` for the same transaction. The `before_commit()` procedure is called before the lock-free reservation commits. The application is expected to implement this method and the applications that use lock-free reservations can choose whether to implement the `before_commit()` or `after_commit()` procedure.

`after_commit` Procedure

Syntax: `procedure after_commit(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `after_commit` procedure is invoked after executing the Saga commit on the current participant when receiving a Saga message with opcode `COMMIT` for the same transaction. The `after_commit` procedure is called after the lock-free reservation commits. The application is expected to implement this method and the applications that use lock-free reservations can choose whether to implement the `before_commit()` or `after_commit()` procedure.

`before_rollback` Procedure

Syntax: `procedure before_rollback(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Description: The `before_rollback` procedure is invoked before executing the Saga rollback on the current participant when receiving a Saga message with opcode `ROLLBACK` for the same transaction. The `before_rollback` procedure is called before the lock-free reservation rollbacks. The application is expected to implement this procedure and the applications that use lock-free reservations can choose whether to implement the `before_rollback` or `after_rollback` procedure.

`after_rollback` Procedure

Syntax: `procedure after_rollback(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `after_rollback` procedure is invoked after executing the Saga rollback on the current participant when receiving a Saga message with opcode `ROLLBACK` for the same transaction. The `after_rollback` procedure is called after the lock-free reservation rollbacks. The application is expected to implement this procedure and the applications that use lock-free reservations can choose whether to implement the `before_rollback` or `after_rollback` procedure.

`forget` Procedure

Syntax: `procedure forget(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The Saga coordinator invokes the `forget` procedure to ask a participant to mark the Saga as forgotten. A Saga marked as forgotten cannot be finalized and requires administrative intervention. The `forget` procedure is triggered upon receiving a negative acknowledgment from the coordinator. A coordinator can respond with a negative acknowledgment to join requests. For example, if a Saga is set with a timeout and a participant join request is received after the timeout, a negative acknowledgment is sent to the participant. Another example of a negative acknowledgment is when a negative acknowledgment is sent on a join request made after a Saga has already been finalized by its initiator.

`is_join` Procedure

Syntax: `procedure is_join (saga_id IN saga_id_t, saga_sender IN VARCHAR2);`

Description: The Saga participants invoke the `is_join` procedure to disassociate themselves from the ongoing Sagas. The procedure gives the participant a choice to move ahead in this Saga or send a `REJECT` back to the initiator.

`reject` Procedure

Syntax: `procedure reject (saga_id IN saga_id_t, saga_sender IN VARCHAR2);`

Description: An initiator invokes the `reject` procedure and receives a notification if a participant rejected the invitation to join the Saga. A participant can return `FALSE` in their `is_join` method, which sends a `REJECT` message to the initiator.

`after_saga` Procedure

Syntax: `procedure after_saga(saga_id in RAW, status in varchar2)`

Description: For PL/SQL-based initiators, the `after_saga` procedure is invoked if defined in the callback package. For PL/SQL-based participants, the `after_saga` method is invoked if defined in the callback package and when event 10855 is set at level 512.

The argument `status` has the stringified values for `enum(org.eclipse.microprofile.lra.annotation.LRAStatus)`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> Along with the `payload`, the implemented methods supply `saga_Id` and the `saga_sender` (sender of this message) as arguments. The clients can use the `saga_Id`, `saga_sender`, and `payload` parameters for book-keeping, maintaining internal states, or both.

28.9.2 Integration with Lock-Free Reservation

Lock-free reservation enables automatic compensation (rollback) of changes to reservable columns. The reservation journal table records an entry for every update to a reservable column in the database. These entries provide the information needed for compensating changes to the reservable columns. The reservation journal entries associated with Saga-related changes to the reservable columns are retained until the Saga is finalized.

The Saga framework uses the `saga_finalization$` dictionary table to track the set of reservable tables and their corresponding journals affected by a Saga. The creation of the reservation journal entry happens during the SQL update. For example, here is a sequence of updates and the respective `saga_finalization$` entries:

```
Update hotel_rooms set rooms = rooms – 2 where date=to_date('20201010');
Update hotel_rooms set rooms = rooms - 1 where date=to_date('20201011');
Update dinner_tables set available_tables= available_tables - 3 where
date=to_date('20201010');
```

Table 28-2 `saga_finalization$` table entries

saga_id	participant	Txn_Id	User#	reservable_table	reservation_journal	status
ABC123	Hotel	1	124	hotel_rooms	SYS_RESERV JRNL_81696	active
ABC123	Hotel	2	124	dinner_tables	SYS_RESERV JRNL_81697	active

The Saga framework performs compensating actions during a Saga rollback. Reservation journal entries provide the data that is required to take compensatory actions for Saga transactions. The Saga framework sequentially processes the `saga_finalization$` table for a Saga branch and executes the compensatory actions using the reservation journal. `Saga_finalization$` entries are marked "compensated" upon successful Saga compensation. `Saga_finalization$` entries are marked "committed" upon successful Saga commit. After a `saga_finalization$` entry is marked as "compensated" or "committed", no further actions are possible.

After a Saga is finalized (successful commit or compensation), the reservation journal entries corresponding to all reservation journals associated with the Saga are deleted. The `saga_finalization$` entries for the given Saga are also deleted.

> **See Also**
>
> Lock-Free Reservation

Copyright © 1999, 2025, Oracle and/or its affiliates.

28.10 AfterSaga Callbacks

An AfterSaga (or AfterLRA) callback method is called after an LRA (Long Running Action, as in a Saga) has completed. An AfterLRA method is used to notify a Saga's participants that the Saga is complete.

For a Saga in a Java-based or PL/SQL application, users can define the AfterLRA callback method. When defined, the callback method is invoked after all the participants of the Saga enter a final state; which is when all the participants involved in the Saga have successfully committed or rolled back the Saga. The Saga coordinator invokes the AfterLRA method on the initiator and all the participants, thereby triggering the `@afterLRA` annotated method for Java-based applications and the afterLRA() method for PL/SQL-based applications.

> **Note**
>
> The initiator's AfterSaga callback is invoked by default. Participant's AfterSaga callback is only invoked when event 10855 is set at level 512.

`@afterLRA` Method

For Java-based initiators and participants, the method decorated with `@afterLRA` is triggered. The method decorated with the `@afterLRA` annotation is expected to have the following signature:

```
@AfterLRA
public void testAfterLRA(SagaMessageContext ctx, LRAStatus status) {
}
```

`SagaMessageContext` is the Saga message context object and `LRAStatus` is defined by LRAStatus (as defined by Eclipse Microprofile).

`after_saga` Method

For PL/SQL-based initiators and participants, the following method is invoked, if defined in the callback package:

```
procedure after_saga(saga_id in RAW, status in varchar2)
```

`status` has the stringified values for `enum(org.eclipse.microprofile.lra.annotation.LRAStatus)`.

The Saga coordinator attempts to invoke the AfterLRA method for all completed Sagas in an interval of 5 minutes (300 seconds). You can modify this interval using the underscore parameter: `_saga_afterlra_interval`.

Dictionary Views

Calling the user-defined AfterLRA method changes the initiator's status in the `USER_SAGAS` dictionary views to:

- `committing` or `rolling back` after the initiator submits `commit` or `rollback`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `committed` or `rolled back` after processing the `commit` or `rollback` for a participant and after all the participants complete `commit` or `rollback` for the initiator.

> **See Also**
>
> USER_SAGAS in *Database Reference Guide* for views-related information

Copyright © 1999, 2025, Oracle and/or its affiliates.

29

Using Lock-Free Reservation

This chapter explains how to use lock-free reservation in database applications.

Topics:

- About Concurrency in Transaction Processing
- Lock-Free Reservation Terminology
- Lock-Free Reservation
- Benefits of Using Lock-Free Reservation
- Guidelines and Restrictions for Lock-Free Reservation

29.1 About Concurrency in Transaction Processing

Transaction processing usually involves flat transactions with data updates that replace old values with new. Most applications store numeric and aggregate values, such as quantity-on-hand of products, bank account balances, stocks, and number of seats available. These numeric aggregate data, although referenced as a single entity, are based on one or more same or similar data. Such data involve subtraction or addition of the values rather than an assignment of the form “data ← value.” For example, a bank account balance is based on the debit and credit transactions that happen on the account.

Applications that provide inventory control, supply chain, financial or investment banking, stocks, travel, and entertainment operate on numeric aggregate data. A numeric aggregate data is generally identified as a “hot” resource because applications continuously and repeatedly read or update such data. The likelihood of many transactions concurrently accessing such data is high. In conventional locking protocols, such as two-phase locking (2PL), execution of concurrent transactions is serialized after a transaction initiates an update on a row and locks it. The initiated update is complete only when the transaction finalizes with a commit or a rollback. Serialization blocks other concurrent transactions from accessing the row until the completion of the transaction that has locked the row. If you allow concurrent transactions to access data, you must control the transactions to preserve application correctness.

In long-running transactions such as microservices applications, a resource remains locked for an extended period, potentially making it a hot resource. The long-term resource locking limits concurrency. In microservices applications that offer services like trip booking services, you may have flight, hotel, and car bookings in long-running transactions. These transactions are typically handled as Sagas and span different services with databases holding locks until the Saga finalization.

Let us look at an example of an online shopping cart to understand how you would handle concurrency in a normal course. A shopping cart has items added to the cart before being sold. Your application must be able to handle multiple transactions at various states like: when a user puts an item into the cart, the item becomes unavailable to other buyers and yet unsold. For concurrent transactions, managing the inventory states and count becomes complex with multiple transactions adding items to the cart and checking out or abandoning the cart. The inventory or similar fields must be locked for a transaction much before a commit or rollback can change the quantity. Locking data for long periods prevents other concurrent transactions

Copyright © 1999, 2025, Oracle and/or its affiliates.

from accessing the item, until the lock is released. Therefore, aiming for isolation and allowing the application to run transactions independently limits concurrency.

For many business applications, blocking concurrent accesses to “hot” data can severely impact performance and reduce user experience and throughput. Applications can benefit if there is improved concurrency coupled with reduced isolation, while also maintaining the atomicity, consistency, and durability properties of transactions. To improve concurrency, it is important to capture the state of a hot resource during a transaction lifecycle and enable data locking only when the resource value is being modified.

See Also

- Lock-Free Reservation for more information about how reservable columns are used to control concurrency in transaction processing.

29.2 Lock-Free Reservation Terminology

- Numeric Aggregate Data
- Hot Data or Hot Resource
- Transaction Lifecycle
- Compensation or Compensating Transaction
- Reservable
- Reservable Column
- Reservable Update
- Reservation Journal
- Lock-free Reservation
- Optimistic Locking
- Saga

Numeric Aggregate Data

Data that involves subtraction (consume, decrement) or addition (replenish, increment) of the quantity (numeric data) rather than an assignment of the form “data ← value”. Operations on numeric aggregate data are commutative in nature.

Hot Data or Hot Resource

Data or a resource that receives high traffic from transactions requiring frequent reads and updates or a resource that is read or updated in a long-running transaction.

Transaction Lifecycle

The states in a business transaction as it transitions from its creation to its committed or rollback state.

Compensation or Compensating Transaction

A compensation or compensating transaction compensates (rolls back) for the already committed transactions of a Saga if any other transaction that is also a part of the Saga fails.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Reservable

Reservable is a column property that you can define for a column that has a numeric data type. A reservable column keeps journals of modification made to the column. These journals are used for concurrency control and auto-compensation.

Reservable Column

A column that contains a hot resource and is identified for lock-free reservation. These columns are declared with a `reservable` column property keyword.

Reservable Update

Numeric aggregate data updates made on a reservable column.

Reservation Journal

A reservation journal is a table associated with the user table that records the modification (increase or decrease by a delta amount with respect to the current value) in the reservable column.

Lock-free Reservation

Any reservable update is treated as a lock-free reservation, implying that the transactions making the updates to a qualifying row do not lock the row, but indicate their intention to modify the reservable column value by a delta amount. The modify operation is recorded in a reservation journal. Lock-free reservations are transformed to actual updates at the commit of the transaction.

Lock-free reservation ensures that the lock is obtained only at the time of commit to update a reservable column. Lock-free reservation is used for hot (high-concurrency) data and are suited for microservices transaction models because of its implicit compensation support.

Optimistic Locking

Optimistic locking is a concurrency control method that allows transactions to use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data that it last read.

Saga

A Saga encapsulates a long running business transaction that is composed of several independent microservices. Each microservice comprises of one or more local transactions that are part of the same Saga.

29.3 Lock-Free Reservation

Lock-free reservation provides an in-database infrastructure for transactions operating on reservable columns to:

- Enable concurrent transactions to proceed without being blocked on updates made to reservable columns
- Issue automatic compensations for reservable updates of successful transactions in a canceled Saga

Here's how lock-free reservation enables applications to include concurrency and auto-compensating features into their transactions:

Copyright © 1999, 2025, Oracle and/or its affiliates.

Declaring a Reservable Column

You can use the `reservable` keyword to declare a reservable column when you create or alter a table. Lock-free reservation is offered on columns with numeric data types. To identify a potential reservable column, look for hot resources with numeric aggregate data that could benefit from improved concurrency.

Reserving a Transaction Update

- When a transaction issues an update operation on a reservable column, the reservable update is placed as a lock-free reservation in a reservation journal. All updates issued on a reservable column are treated as lock-free reservation.
- The transaction update does not lock the row (that has the reservable update) but indicates its intention to modify (add or subtract) the reservable column in the row by a delta amount. The modification amount is reserved and promised so that the transaction may proceed without waiting on other uncommitted, concurrent transactions that have made earlier reservations on the same row's reservable column. The reservation enables other concurrent transactions to issue reservable updates to the same row.
- The operation of modification (increase or decrease by the delta amount with respect to the current value of the reservable column) is recorded in a reservation journal. Instead of reading and writing the actual value of the reservable column, transactions issue operations to increment or decrement the reservable column value. Update requests made to the reservable column are recorded in a reservation journal.

Verifying and Deferring an Update

The transaction, based on the constraints placed on the reservable column, decides whether the quantity is sufficient to make the update. The transaction checks for the following:

- Verifies that the update can succeed. The update to the reservation journal is allowed to proceed if there is a sufficient balance. If the balance is not sufficient to fulfill the update (consume) request, the update request to the reservable column is failed, without relying on any active (not yet committed) replenishment to succeed.
- Checks for any constraints or bounds (`CHECK` constraints) on reservable columns to enforce business rules and to ensure application correctness. `CHECK` constraints can include checks for reservable and non-reservable columns.
- Defers the actual update to reservable column until the commit time to improve concurrency

Auto-compensating Successful Updates in a Failed Transaction

Lock-free reservation enables the tracking of the state transitions of a reservable column through its transaction lifecycle. If a transaction is canceled (after partial completions) due to reasons such as insufficient balance or cancellation of a Saga, lock-free reservation issues automatic compensation or rollback of the reservable updates.

Ensuring Durability of a Reservable Update

The lock-free reservations are transformed to the actual updates at the commit of the transaction. Rollback of the transaction voids all lock-free reservations that the transaction holds in the reservation journals. Rollback to a savepoint removes the lock-free reservations made by the transaction after the affected savepoint.

For Sagas in microservices, lock-free reservations enable automatic compensation of the reservable updates made for successful local transactions in an canceled Saga. Lock-free

Copyright © 1999, 2025, Oracle and/or its affiliates.

reservation enables tracking the reservable updates within the database during the execution of transactions and retains the journals beyond the commit of the transaction until the Saga finalization.

> **Note**
>
> The Lock-Free Reservation feature does not lock rows until commit time, hence Priority Transactions (another 26ai feature) is a no-op for transactions that only do lock-free reservations.

> **See Also**
>
> - Priority Transactions for more information about the Priority Transactions feature

29.3.1 Comparing Optimistic Locking and Lock-Free Reservation

Lock-free reservation guarantees the outstanding reservations by operating within bounds as follows:

- Journals on-going requests.
- Introduces new reservation journal columns to track projected values based on the pending requests.
- Consults the journal to allow or reject new requests based on outstanding reservations.
- Rejects new requests that cannot be satisfied due to outstanding reservations or if the request quantity is greater than the availability.

In comparison, optimistic locking does not track or guarantee reservations.

Here are other drawbacks of the optimistic locking approach:

- Requests may not be satisfied at the commit time because of insufficient quantity.
- Possibility of a transaction getting canceled at the end. If a long-running transaction is canceled, work is wasted and the changes must be rolled back.

29.3.2 Creating a Reservable Column at Table Creation

When you create a new table, you can declare a reservable column using the `reservable` keyword.

1. Modify the `CREATE TABLE` command to enable lock-free reservation as follows:

```
Create_table_command::={relational_table | object_table | XMLType_table }
Relational_table::= CREATE TABLE [ schema. ] table …;
relational_properties::= [ column_definition ]
column_definition::= column_name datatype reservable [CONSTRAINT constraint_name
check_constraint]

CREATE TABLE Account( ID NUMBER PRIMARY KEY,
 Name VARCHAR2(10),
 Balance NUMBER reservable CONSTRAINT minimum_balance CHECK (Balance >= 50))
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The creation of the "Account" table with the reservable column "Balance" creates an associated reservation journal table. The reservation journal table is created under the same user schema and in the same tablespace as the user table. The reservation journal table also has deferred segment creation enabled.

> **Note**
>
> The `CREATE TABLE` statement syntax supports the `RESERVABLE` keyword but not the `NOT RESERVABLE` keyword. The `NOT RESERVABLE` keyword is supported in the `ALTER TABLE` command.

> **See Also**
>
> Reservation Journal Table Columns for more information about the columns used in the reservation journal table.

2. Use the `ALL_CONSTRAINTS` view to get the constraint details.

```
SELECT table_name, constraint_name, search_condition
 FROM ALL_CONSTRAINTS
 WHERE table_name='ACCOUNT';
```

29.3.2.1 Reservation Journal Table Columns

A reservation journal table contains the following column information.

The `DESCRIBE` statement gives you the actual names of the columns in the reservation journal table associated with the earlier mentioned Account table:

```
SQL>desc SYS_RESERVJRNL_<object_number_of_base_table>;
```

Table 29-1 Reservation Table Columns

Name	Null?	Type	Description
ORA_SAGA_ID$		RAW(16)	Saga ID of the transaction (0 for non-saga transactions)
ORA_TXN_ID$		RAW(8)	Transaction ID of the transaction (containing usn, slot, seq)
ORA_STATUS$		VARCHAR2(11)	Status of the Txn ID, with values: {ACTIVE, COMMITTED, COMPENSATED}
ORA_STMT_TYPE$		VARCHAR2(6)	DML statement type {UPDATE}
ID	NOT NULL	NUMBER	Primary Key column of user table

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 29-1 (Cont.) Reservation Table Columns

Name	Null?	Type	Description
BALANCE_OP		VARCHAR2(1)	Reservable column operation with operations having '+' or '-' for replenishment or consumption
BALANCE_RESERVED		NUMBER	Reservable column reserved and is the amount reserved from the reservable column

29.3.3 Adding or Modifying Reservable Columns

You can modify the `ALTER TABLE` statement to add a reservable column or change a reservable column into a non-reservable column.

1. To add a reservable column, modify the `ALTER TABLE` command as follows:

```
ALTER TABLE [ schema.]table
 [add [column_definition]]…;
 column_definition::= column_name datatype reservable [default <value>]
 [CONSTRAINT constraint_name check_constraint]

ALTER TABLE Account
 ADD (Balance NUMBER reservable default 50 CONSTRAINT minimum_balance CHECK (Balance
>= 50));
```

2. To change an existing column to a reservable column, modify the `ALTER TABLE` command as follows:

```
ALTER TABLE [ schema.]table
 [modify [column_definition]]…;
 column_definition::= column_name reservable [default <value>]
 [CONSTRAINT constraint_name check_constraint]
```

   To change an existing QOH column to a reservable column and to optionally add a new constraint:

```
ALTER TABLE PRODUCTS
MODIFY (QOH reservable default 0 CONSTRAINT maxAmount CHECK (QOH <= 100));
```

3. To change a reservable column to a non-reservable column, modify the `ALTER TABLE` command as follows:

```
ALTER TABLE [ schema.]table
 [modify [column_definition]]…;
 column_definition::= column_name not reservable]
```

   To change an existing reservable column QOH to a non-reservable column:

```
ALTER TABLE PRODUCTS modify (QOH not reservable);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> A reservable column can be converted to a non-reservable column using the earlier mentioned `ALTER TABLE` command. Although the lock-free reservations for the column are disabled, once a reservable column is changed to a non-reservable column, applications may choose to enforce the constraints. Hence, the constraints are not automatically dropped when a column is converted to a non-reservable column. You can choose to drop the constraints using the `ALTER TABLE <table_name> DROP CONSTRAINT <constraint_name>` statement.

29.3.4 About CHECK Constraints in Reservable Columns

A `CHECK` constraint lets you specify a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either `TRUE` or unknown (due to a `NULL`). When Oracle evaluates a check constraint condition for a particular row, any column names in the condition refer to the column values in that row.

> **Note**
>
> Oracle does not verify that conditions of check constraints are not mutually exclusive. Therefore, if you create multiple check constraints for a column, design them carefully so their purposes do not conflict. Do not assume any particular order of evaluation of the conditions.

Table-level CHECK Constraints for Reservable Columns

You can have reservable columns in table-level `CHECK` constraints. If a table-level `CHECK` constraint that involves reservable and non-reservable columns fails validation at commit time, then the transaction is canceled. The cancel happens because the non-reservable column values cannot be guaranteed using the reservation mechanism.

For example:

```
CREATE TABLE Account(
 ID NUMBER PRIMARY KEY,
 Name VARCHAR2(10),
 Balance NUMBER reservable,
 Earmark NUMBER,
 Limit NUMBER,
 CONSTRAINT minimum_balance CHECK (Balance + Limit – Earmark >= 0))
```

You can also define reservable columns without constraints. For such reservable columns all lock-free reservations are successful.

No storage clause specification is supported for reservable columns.

Copyright © 1999, 2025, Oracle and/or its affiliates.

29.3.5 Example: Conventional Locking and Lock-Free Reservation

The examples below provide a purchase transaction in conventional and lock-free reservation modes.

Conventional Locking (with Long-held Locks)

The following example uses traditional locking to allow a purchase of a $25 item while maintaining a $50 minimum balance:

- A `SELECT FOR UPDATE` is first issued to read and lock the balance.
- If the balance is at least 75, the item purchase is allowed.
- The `UPDATE` then debits the balance.
- The transaction then commits.
- An insufficient balance causes a canceling of the transaction.

```
CREATE TABLE Account (
 ID NUMBER PRIMARY KEY,
 Name VARCHAR2(10),
 Balance NUMBER CONSTRAINT minimum_balance CHECK (Balance >= 50));

DECLARE current NUMBER;

BEGIN
 -- Read and Lock account balance
 SELECT Balance INTO current
 FROM Account
 WHERE ID = 12345 FOR UPDATE;

 IF current >= 75 THEN
  -- Sufficient funds: Perform item purchase
  PurchaseItem();
  -- Debit account balance and commit
  UPDATE Account
  SET Balance = Balance - 25
  WHERE ID = 12345;
  COMMIT;
 ELSE
  ROLLBACK; -- Insufficient funds, so cancel
 END IF;
END;
```

Lock-Free Reservation (with Short-held Locks)

The following example uses lock-free reservation to allow a purchase of a $25 item while maintaining a $50 minimum balance. The reservable column constraint allows a reservation to be placed on a column value without locking the row.

- The balance update reserves $25 without locking the account.
- If the reservation succeeds, the item purchase is allowed to proceed.
- The final commit locks the account row and applies the balance debit of $25 as recorded in the reservation.
- If the reservation fails due to insufficient funds, the update statement fails with the `CHECK` constraint violation.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CREATE Table Account(
 ID NUMBER PRIMARY KEY,
 Name VARCHAR2(10),
 Balance NUMBER RESERVABLE CONSTRAINT minimum_balance CHECK (Balance >= 50));

BEGIN
 -- Reserve 25 from account balance
 UPDATE Account SET Balance = Balance - 25
 WHERE ID = 12345;
 -- If reservation succeeds perform item purchase
 PurchaseItem();
 -- The commit finalizes the balance update
 COMMIT; -- This gets the account row lock
 EXCEPTION WHEN Check_Constraint_Violated
 -- This indicates that the reservation failed
 THEN ROLLBACK;
END;
```

29.3.6 Querying Reservable Column Views

You can run queries on the dictionary views of reservable columns to get information about reservable columns.

You can run queries on the `DBA_TAB_COLUMNS`, `USER_TAB_COLUMNS` and `ALL_TAB_COLUMNS` views to check if a column is declared as a reservable column.

```
SELECT table_name, column_name , reservable_column
 FROM user_tab_columns
 WHERE table_name = <table name>;
```

You can run queries on the `DBA_TAB_COLS`, `USER_TAB_COLS`, and `ALL_TAB_COLS` views to check if a column is declared as a reservable column.

```
SELECT table_name, column_name , reservable_column
 FROM user_tab_cols
 WHERE table_name = <table name>;
```

Example Query:

```
SQL> SELECT table_name, column_name , reservable_column
 FROM user_tab_cols
 WHERE table_name = 'ACCOUNT';
```

Result:

```
TABLE_NAME     COLUMN_NAME    RES
-----------------------------------
ACCOUNT         NAME          NO
ACCOUNT         BALANCE       YES
ACCOUNT         ID            NO

3 rows selected
```

You can run queries on the `DBA_TABLES`, `USER_TABLES`, and `ALL_TABLES` views to check if the user table has one or more reservable columns.

```
SELECT table_name, has_reservable_column
 FROM user_tables
 WHERE table_name = <table name>;
```

Example Query:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
SQL> SELECT table_name, has_reservable_column
 FROM user_tables
 WHERE table_name = 'ACCOUNT';
```

Result:

```
TABLE_NAME     HAS
--------------------
ACCOUNT        YES

1 row selected
```

29.4 Benefits of Using Lock-Free Reservation

Improved User Experience and Concurrency

Lock-free reservation holds locks on hot data for short intervals of time, thereby improving user experience and concurrency. Transactions reserve an amount from a reservable column value without locking it. The locking happens only when the value is modified during the commit of the transaction.

Automatic Compensation

When using compensating functions, you need to track the dependencies to be able to transition the database into a consistent state. For Sagas, these dependencies need to be tracked for a long period (until the change is confirmed). You can use lock-free reservation in your Saga implementations. With lock-free reservation, if a Saga transaction is canceled, implicit compensating transactions are automatically issued to take care of the rollback of other transactions of the Saga that have already committed. You do not need to go through the complexity of writing compensating functions. Lock-free reservation enables auto-compensation through the journals to reverse any successful update on failed Sagas.

Efficient Resource Usage

When you use lock-free reservation, multiple transactions can run in parallel and use resources without blocking each other out. Efficiency in resources usage improves with less waiting and response times.

Broader Scope

Reservable updates are done on numeric aggregate data, which is integral to many applications that operate on a wide variety of data. Improved concurrency using lock-free reservation can benefit applications wherever there are a high rate of rows with reservable updates. Applications that have reservable updates in long-running transactions can benefit the most from improved concurrency in transactions. Some of these applications are those that deal in banking, inventory control, ticketing, and event reservation.

29.5 Guidelines and Restrictions for Lock-Free Reservation

This section provides the guidelines and restrictions for lock-free reservation.

29.5.1 Guidelines and Restrictions for Reservable Columns

When using lock-free reservation, follow these guidelines for reservable columns and user tables with reservable columns:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- The Schema definition of user tables declare the reservable columns with the `reservable` keyword. Reservable columns provide lock-free reservations.
- A reservable column can be specified for Oracle numeric data type (`NUMBER`, `INTEGER`, and `FLOAT`) columns only.
- A reservable column cannot be a `Primary Key` or an identity column (or virtual column) because the reservable column is an aggregate type.
- A user table can have at most ten reservable columns.
- User tables that have reservable columns must have a `Primary Key`.
- Indexes are not supported on reservable columns.
- Composite reservable columns are not allowed.
- Reservable columns are not allowed in Block Chain tables and Sharded tables.
- Reservable columns can be included only in `CHECK` constraints expression. The `CHECK` constraint can be at the column-level or table-level. User-defined operational constraints are used for reservable columns to ensure application correctness.
  - Optional `CHECK` constraints can be specified on the reservable columns.
  - Reservable columns cannot be involved in any other types of constraints. Non-`CHECK` constraints involving the reservable columns are not allowed. Reservable columns cannot be part of a foreign key constraint.
- External, Cluster, IOT and temporary tables cannot have reservable columns.
- Partitioning cannot be made on reservable columns.
- Two-phase online DDL optimization is not provided for user tables with reservable columns.
- Dropping a reservable column from a user table removes the corresponding lock-free reservation tracking columns from the reservation journal table. If the last reservable column is removed from a user table, the reservation journal table is dropped. Transactions with pending reservations must finalize before you can drop the reservable column or mark the column as `UNUSED`.

29.5.2 Guidelines and Restrictions for Update Statements

When using lock-free reservation, follow these guidelines for the `UPDATE` statements:

- Updates to reservable columns must be specified as one of the following:

```
UPDATE <table_name>
SET <reservable_column_name> = <reservable_column_name> + (<expression>)
WHERE <primary_key_column> = <expression>

UPDATE <table_name>
SET <reservable_column_name> = <reservable_column_name> - (<expression>)
WHERE <primary_key_column> = <expression>
```

> **Note**
>
> Do not use `SET <reservable_column_name> = <value>`. Direct assignment of a value raises an error.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> For composite primary keys, all the primary key columns must be specified in the `WHERE` clause of the reservable update.

- Multiple reservable columns of a table can be updated in a single update statement.
- Mixing reservable and non-reservable column updates in the same update statement is not allowed. Also, DML returning clause is not supported in reservable column update statements.
- Updates to reservable columns do not lock the row until the commit of the transaction. Instead, reservable columns provide for lock-free reservations. Lock-free reservations allow other concurrent transactions updating the reservable columns without being blocked.
- A reservable update issued on a locked row (locked by either the same or a different transaction that has updated a non-reservable column of the row) is allowed to obtain lock-free reservation.
- The updates that are transformed to pending reservations on the reservable columns ensure that the constraints on the reservable columns are not violated after considering the pending reservations.
- A transaction can read its own lock-free reservations by selecting from the reservation journal tables (associated with the user tables) on which the transaction has issued a reservable update. Reservations made by other transactions are not visible.

29.5.3 Guidelines for Inserts and Deletes

When using lock-free reservation, follow these guidelines for the `INSERT` and `DELETE` statements issued on tables with reservable columns:

- Transactions can insert an entire row with values for the reservable columns without any change in behavior.
- The inserted row is not visible to other transactions for any reservable updates until the inserting transaction commits.
- If a `DELETE` statement is issued when there are pending reservations for the rows of a user table that are to be deleted, the transactions with active lock-free reservations against those rows should complete before the delete can proceed. The `DELETE` statement is internally retried for an interval of 5 seconds and is allowed if there are no more pending reservations for the affected rows. After the timeout period, if the `DELETE` statement could not proceed, an error with resource busy message is raised, and the `DELETE` statement can be issued at a later time.

29.5.4 Guidelines for Concurrent DDL Statements

When using lock-free reservation, follow these guidelines for the concurrent DDL statements issued on tables with reservable columns:

- If a DDL statement is in progress on a table that has a reservable column, no reservable updates can operate on the user table until the DDL completes.
- If a DDL statement is issued when there are pending reservations for any rows of a user table, the DDL statement is blocked until the transactions with active reservations complete.

Copyright © 1999, 2025, Oracle and/or its affiliates.

29.5.5 Restrictions for Reservation Journal Table

Observe these restrictions for reservation journal table:

User DML and DDL operations are not permitted on a reservation journal table. You cannot create or modify a reservation journal table using DML. You cannot use SQL to drop, truncate, rename, or change the reservation table's definition.

Copyright © 1999, 2025, Oracle and/or its affiliates.

30

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 the 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

30.1 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

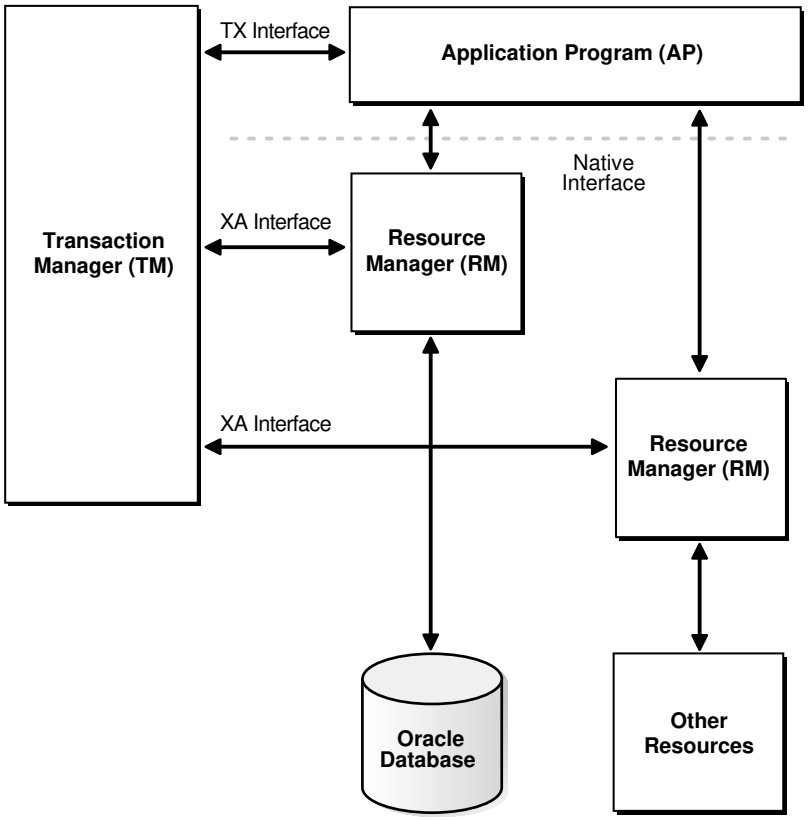
Copyright © 1999, 2025, Oracle and/or its affiliates.

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 30-1 illustrates a possible X/Open DTP model.

Figure 30-1 Possible DTP Model



Topics:

- DTP Terminology
- Required Public Information

30.1.1 DTP Terminology

- Resource Manager (RM)
- Distributed Transaction
- Branch
- Transaction Manager (TM)

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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", 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.1.2 Required Public Information

As a resource manager, Oracle Database must publish the information described in Table 30-1.

Table 30-1 Required XA Features Published by Oracle Database

XA Feature	Oracle Database Details
`xa_switch_t` structures	The Oracle Database `xa_switch_t` structure name is `xaosw` for static registration and `xaoswd` for dynamic registration. These structures contain entry points and other information for the resource manager.
`xa_switch_t` resource manager	The Oracle Database resource manager name within the `xa_switch_t` structure is `Oracle_XA`.
Close string	The close string used by `xa_close` is ignored and can be null.
Open string	For the description of the format of the open string that `xa_open` uses, see Defining the xa_open 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 Precompilers (such as Pro*C/C++, Pro*COBOL, and others), then link with `$ORACLE_HOME/rdbms/lib/xaonsl.o` or `$ORACLE_HOME/rdbms/lib32/xaonsl.o` (for 32 bit application on 64 bit platforms).
Requirements	None. The functionality to support XA is part of both Standard Edition and Enterprise Edition.

30.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.2.1 Oracle XA Library Subprograms

XA Library subprograms are described in Table 30-2.

Table 30-2 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 30-2 except to understand the role played by the `xa_open` string.

30.2.2 Oracle XA Interface Extensions

Oracle Database's XA interface includes some additional functions, which are described in Table 30-3.

Table 30-3 Oracle XA Interface Extensions

Function	Description
`OCISvcCtx *xaoSvcCtx(text *dbname)`	Returns the OCI service handle for a given XA connection. The `dbname` parameter must be the same as the `DB` parameter passed in the `xa_open` string. OCI applications can use this routing instead of the `sqlld2` calls to obtain the connection handle. Hence, OCI applications need not link with the `sqllib` library. The service handle can be converted to the Version 7 OCI logon data area (LDA) by using `OCISvcCtxToLda` [Version 8 OCI]. Client applications must remember to convert the Version 7 LDA to a service handle by using `OCILdaToSvcCtx` after completing the OCI calls.
`OCIEnv *xaoEnv(text *dbname)`	Returns the OCI environment handle for a given XA connection. The `dbname` parameter must be the same as the `DB` parameter passed in the `xa_open` string.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 30-3 (Cont.) Oracle XA Interface Extensions

Function	Description
`int xaosterr(OCISvcCtx *SvcCtx,sb4 error)`	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 `xa_start` failed. The function returns XA_OK if the error was not generated by the XA module or a valid XA error if the error was generated by the XA module.

30.3 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

30.3.1 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.
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 they 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`.
3. Using the open string information, install the RM into the TPM configuration. Follow the TPM vendor instructions.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.
5. Start the relevant database instances to bring Oracle XA applications on-line. Perform this task before starting any TPM servers.

> **See Also**
>
> - Defining the xa_open String for information about how to define open string, and specify an Oracle System Identifier (SID) or a trace directory that is different from the defaults
> - Your Oracle Database platform-specific documentation for the location of the `catxpend.sql` script

30.3.2 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 Defining the xa_open String..
2. Develop the applications.

   Observe special restrictions on transaction-oriented SQL statements for precompilers.

   > **See Also**
   >
   > Developing and Installing XA Applications

3. Link the application according to TPM vendor instructions.

30.3.3 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

30.3.3.1 Syntax of the xa_open String

You can define an open string with the syntax shown in Example 30-1.

These strings shows sample parameter settings:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

Example 30-1 xa_open String

```
ORACLE_XA{+required_fields...} [+optional_fields...]
```

30.3.3.2 Required Fields for the xa_open String

The *required_fields* placeholder in Example 30-1 refers to any of the name-value pairs described in Table 30-4.

Table 30-4 Required Fields of xa_open string

Syntax Element	Description
`Acc=P//`	Specifies that no explicit user or password information is provided and that the operating system authentication form is used.
`Acc=P/user/password`	Specifies the user name and password for a valid Oracle Database account. As described in DBA or System Administrator Responsibilities, ensure that HR has the `READ` or `SELECT` privilege on the `DBA_PENDING_TRANSACTIONS` table. The password accepts all special characters (including =) except the following characters: +, /, \, and £.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 30-4 (Cont.) Required Fields of xa_open string

Syntax Element	Description
`SesTm=`*`session_time_limit`*	Specifies the maximum number of seconds allowed in a transaction between one service and the next, or between a service and the commit or rollback of the transaction, before the system terminates the transaction. For example, `SesTM=15` indicates that the session idle time limit is 15 seconds. For example, if the TPM uses remote subprogram calls between the client and the servers, then `SesTM` applies to the time between the completion of one RPC and the initiation of the next RPC, or the `tx_commit`, or the `tx_rollback`. The value of `0` indicates no limit. Entering a value of `0` is strongly discouraged. It might tie up resources for a long time if something goes wrong. **Note** If an XA session has specified `SesTM=0`, then even after `xa_end`, if the session does an `xa_close` or otherwise terminates the connection, then the transaction branch is timed out as if the session had ended while still attached to the branch.

See Also

Oracle Database Administrator's Guide for more information about administrator authentication

30.3.3.3 Optional Fields for the xa_open String

The *`optional_fields`* placeholder in Example 30-1 refers to any of the name-value pairs described in Table 30-5.

Table 30-5 Optional Fields in the xa_open String

Syntax Element	Description
`NoLocal= true | false`	Specifies whether local transactions are allowed. The default value is `false`. If the application must disallow local transactions, then set the value to `true`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 30-5 (Cont.) Optional Fields in the xa_open String

Syntax Element	Description
DB=*db_name*	Specifies the name used by Oracle Database precompilers to identify the database. For example, DB=payroll specifies that the database name is payroll and that the application server program uses that name in AT clauses. Application programs that use only the default database for the Oracle Database precompiler (that is, they do not use the AT clause in their SQL statements) must omit the DB=*db_name* clause in the open string. Applications that use explicitly named databases must indicate that database name in their DB=*db_name* field. Oracle Database Version 7 OCI programs must call the sqlld2 function to obtain the correct context for logon data area (Lda_Def), which is the equivalent of an OCI service context. Version 8 and higher OCI programs must call the xaoSvcCtx function to get the OCISvcCtx service context. The *db_name* is not the SID and is not used to locate the database to be opened. Rather, it correlates the database opened by this open string with the name used in the application program to run SQL statements. The SID is set from either the environment variable ORACLE_SID of the TPM application server or the SID given in the Oracle Net clause in the open string. The Oracle Net clause is described later in this section.Some TPM vendors provide a way to name a group of servers that use the same open string. You might find it convenient to choose the same name both for that purpose and for db_name.
LogDir=*log_dir*	Specifies the path name on the local system where the Oracle XA library error and tracing information is to be logged. The default is $ORACLE_HOME/rdbms/log if ORACLE_HOME is set; otherwise, it specifies the current directory. For example, LogDir=/xa_trace indicates that the logging information is located under the /xa_trace directory. Ensure that the directory exists and the application server can write to it.
Objects= true|false	Specifies whether the application is initialized in object mode. The default value is false. If the application must use certain API calls that require object mode, such as OCIRawAssignBytes, then set the value to true.
MaxCur=*maximum_#_of_open_cursors*	Specifies the number of cursors to be allocated when the database is opened. It serves the same purpose as the precompiler option maxopencursors. For example, MaxCur=5 indicates that the precompiler tries to keep five open cursors cached. This parameter overrides the precompiler option maxopencursors that you might have specified in your source code or at compile time.
SqlNet=*db_link*	Specifies the Oracle Net database link to use to log on to the system. This string must be an entry in tnsnames.ora. For example, the string SqlNet=inst1_disp might connect to a shared server at instance 1 if so defined in tnsnames.ora. You can use the SqlNet parameter to specify the ORACLE_SID in cases where you cannot control the server environment variable. You must also use it when the server 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 SqlNet=localsid1, where localsid1 is an alias defined in the tnsnames.ora file.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 30-5 (Cont.) Optional Fields in the xa_open String

Syntax Element	Description
`Loose_Coupling=true | false`	Specifies whether locks are shared. Oracle Database transaction branches within the same global transaction can be coupled tightly or loosely. If branches are loosely coupled, then they do not share locks. Set the value to `true` for loosely coupled branches. If branches are tightly coupled, then they share locks. Set the value to `false` for tightly coupled branches. The default value is `false`. **Note:** When running Oracle RAC, if transaction branches land on different Oracle RAC instances, then they are loosely coupled even if `Loose_Coupling=false`.
`SesWt=`*`session_wait_limit`*	Specifies the number of seconds Oracle Database waits for a transaction branch that is being used by another session before XA_RETRY is returned. The default value is 60 seconds.
`Threads=true | false`	Specifies whether the application is multithreaded. The default value is `false`. If the application is multithreaded, then the setting is `true`.
`FAN=true | false`	Specifies whether the application will use Fast Application Notification (FAN). The default value is `false`. If the application will use FAN, then the setting is `true`.

See Also

Oracle Database Administrator's Guide for information about FAN

30.3.4 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++.

30.3.4.1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
EXEC SQL UPDATE Emp_tab SET Sal = Sal*1.5;
```

30.3.4.2 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, the employee's department number (DEPTNO) in another, and the employee's manager in a third database. Configure the open strings in the transaction manager as shown in Example 30-2.

Example 30-2 Sample Open String Configuration

```
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
```

There is no DB=*db_name* field in the last open string in Example 30-2.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.3.5 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 Developer's Guide*

30.3.6 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 30-6, but not the XA Library Subprograms listed in Table 30-2.

The TMs typically control the transactions through the XA interface. This interface includes the functions described in Table 30-2.

Table 30-6 TX Interface Functions

TX Function	Description
`tx_open`	Logs into the resource manager(s)
`tx_close`	Logs out of the resource manager(s)
`tx_begin`	Starts a transaction
`tx_commit`	Commits a transaction
`tx_rollback`	Rolls back the transaction

Most TPM applications use a client/server architecture in which an application client requests services and an application server provides them. The examples shown in "Examples of Precompiler Applications" 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

own local services. As shown in "Examples of Precompiler Applications", 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.

30.3.7 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 30-3 shows a transaction started by an application server.

Example 30-3 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 30-4 shows a transaction started by an application client.

Example 30-4 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>
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  EXEC SQL UPDATE ...;
  ...
  <return service status back to client>
}
```

30.3.8 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 30-7 lists the TPM functions that replace regular Oracle Database statements when migrating precompiler or OCI applications to TPM applications.

Table 30-7 TPM Replacement Statements

Regular Oracle Database Statements	TPM Functions
`CONNECT`*`user/password`*	`tx_open` (possibly implicit)
implicit start of transaction	`tx_begin`
`SQL`	Service that runs the SQL
`COMMIT`	`tx_commit`
`ROLLBACK`	`tx_rollback`
disconnect	`tx_close` (possibly implicit)

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.3.9 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 unusual.

> **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

30.3.9.1 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.

30.3.9.2 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.

30.3.10 Using the DBMS_XA Package

PL/SQL applications can use the Oracle XA library with the `DBMS_XA` package.

Copyright © 1999, 2025, Oracle and/or its affiliates.

In Example 30-5, 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 30-5 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
        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
```

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_XA` package

30.4 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

30.4.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 30-8 explains the meaning of each element.

Table 30-8 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
ORA-01017	Oracle Database information that was returned

Topics:

- xa_open String DbgFl
- Trace File Locations

30.4.1.1 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.

30.4.1.2 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):

Copyright © 1999, 2025, Oracle and/or its affiliates.

- %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.

30.4.2 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.

30.4.3 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_name*xa.oracle.com

Remember that the *db_name* is always specified as DB=*db_name* in the open string. If you do not specify this field in the open string, then the value of this column is NULLxa.oracle.com for transactions generated by Oracle XA applications.

For example, 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.5 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

30.5.1 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.

30.5.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 30-9 summarizes the trade-offs between tightly coupled branches and loosely coupled branches.

Table 30-9 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

30.5.3 Using Oracle XA with Oracle Real Application Clusters (Oracle RAC)

As of Oracle Database 11*g* 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

30.5.3.1 Oracle RAC XA Limitations

Identify and maintain XA affinity while performing transactions on one RAC node, irrespective of the number of connections or operating system processes that are involved in the transaction.

See:
Distributed Transaction Processing in Oracle RAC

30.5.3.2 GLOBAL_TXN_PROCESSES Initialization Parameter

The initialization parameter `GLOBAL_TXN_PROCESSES` specifies the initial number of GTX*n* background processes for each Oracle RAC instance. Its default value is 1.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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`

30.5.3.3 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 11*g* 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 11*g* 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 11*g* 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

30.5.3.4 Managing Instance Recovery in Oracle RAC with DTP Services (10.2)

Before Oracle Database 10*g* 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 10*g* 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 9*g* Release 2 (9.2), `xa_recover` is required to wait for distributed data manipulation language (DML) statements to complete on remote sites.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

30.5.3.5 Global Uniqueness of XIDs in Oracle RAC

Before Oracle Database 11*g* 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 11*g* 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

30.5.3.6 Tight and Loose Coupling

Oracle Database transaction branches within the same global transaction can be coupled either tightly or loosely. Ordinarily, coupling type is determined by the value of the `Loose_Coupling` field of the `xa_open` string (see Table 30-5). However, if transaction branches land on different Oracle RAC instances when running Oracle RAC, they are loosely coupled even if `Loose_Coupling=false`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Real Application Clusters Administration and Deployment Guide* for information about services and distributed transaction processing in Oracle RAC
- Managing Transaction Branches in Oracle XA Applications

30.5.4 SQL-Based Oracle XA Restrictions

This section describes restrictions concerning these SQL operations:

- Rollbacks and Commits
- DDL Statements
- Session State
- EXEC SQL

30.5.4.1 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 `orol`. You can roll back a global transaction by calling `tx_rollback`.

Similarly, a precompiler application must not have the `EXEC SQL COMMIT WORK` statement in the middle of a global transaction. An OCI application must not run `OCITransCommit` or the Version 7 equivalent `ocom`. For example, use `tx_commit` or `tx_rollback` to end a global transaction.

30.5.4.2 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.

30.5.4.3 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.

30.5.4.4 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

30.5.5 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

  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`).

Copyright © 1999, 2025, Oracle and/or its affiliates.

31

Developing Applications with Sessionless Transactions

In this chapter, learn about Sessionless transactions and how to implement them to manage transactional workloads in your applications.

- Introduction to Managing Transactions and Sessionless Transactions
- Sessionless Transactions Overview
- Sessionless Transactions Capabilities
- Benefits of Sessionless Transactions
- Using Sessionless Transactions
- Sessionless Transactions and Oracle Coordinated Distributed Transaction Interoperability
- Restrictions for Sessionless Transactions

31.1 Introduction to Managing Transactions and Sessionless Transactions

Using database resources efficiently is a challenge when you develop applications that use transactional workloads. Typically, managing a transaction requires the connection and session resources to be tied to the transaction throughout the transaction's lifecycle, and therefore the session or connection can be released only after the transaction has ended. Ideally, applications with think time in the application logic would want to grab a session from the database, submit a unit of work, release the session, attend to the application logic, grab another session, resume the transaction, submit another unit of work, and so on, until they commit the transaction in the end. As is evident, the latter approach enables applications to use resources more efficiently because the transaction is not tied to a session or connection, which can then be used by another client. The Sessionless Transactions feature is based on the latter approach. In Sessionless Transactions, after you start a transaction, you have the flexibility to suspend and resume the transaction across sessions during the transaction's lifecycle. Moreover, when you use Sessionless Transactions, the database internally coordinates the two-phase commit (2PC) protocol, without the need for any application-side logic. Also, Sessionless Transactions can be used in a single-instance configuration or multi-instance Real Application Clusters (RAC) configuration.

> **Note**
>
> Sessionless Transactions are applicable only to a single database.

To get a better perspective of how important the Sessionless Transactions feature can be to your application, take a look at how an application would manage transactions with and without Sessionless Transactions.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Managing Transactions without Sessionless Transactions

Without Sessionless Transactions, to manage transactions in your application, you would need to choose between one of the following two approaches:

- Start a transaction on a session in an instance and continue to associate the transaction with the same session and instance until the transaction is finalized.

  Downside: When a transaction is associated with a session and a connection, the session and connection in the instance are held up for the time the transaction is active, even when the resource utilization is very low. With the addition of more and more connections, you risk the chances of the database running out of connections.

- Coordinate the eXtended Architecture (XA) protocol through a transaction manager. A transaction manager is required to coordinate an XA transaction across branches on different database instances and sessions, and also coordinate the 2PC process to commit or roll back the XA transaction.

  Downside: Although the connections are not held up and you can multiplex the connection across clients, you still need the transaction manager infrastructure to coordinate an XA transaction. The transaction manager must work in the middle tier and remember the status of all the branches. Any network failure can result in in-doubt transactions, which can only be resolved using a recovery process. Moreover, an additional mid-tier component like a transaction manager introduces yet another stateful service that has to be managed, and kept highly available - a huge management overhead. If the service fails, then locks are held in the database leading to cascading failures, hangs, and database outages.

There are solutions such as Oracle XA Transactions, which is based on the XA open standard, that allow you to multiplex transactions between connections and sessions. Oracle XA guarantees that in all the participating databases the transactional updates are either committed or fully rolled back. However, to use Oracle XA, you must deploy a transaction manager, such as MicroTX, Tuxedo, or WebLogic as a middle-tier component that resides outside Oracle Database.

There are other disadvantages of using XA Transactions, such as:

- XA transactions can result in high transaction latency owing to slow resource managers or unreliable network between a transaction manager and resource managers.
- XA transactions are only as reliable as the external transaction managers, even though the individual resource managers are highly reliable and available like Oracle Database.
- Transaction managers must create and manage branches across database instances or sessions, thereby requiring more memory space to keep the information on branches and more time to find the correct branch when resuming a transaction on a certain session and instance. Therefore, transaction managers must have the knowledge of individual instances and run the 2PC operation between the branches created across database instances. Any failure can lead to the transaction going in-doubt and blocking subsequent work.

> **See Also**
>
> Developing Applications with Oracle XA for information about Oracle XA

Copyright © 1999, 2025, Oracle and/or its affiliates.

On the contrary, when you use Sessionless Transactions, you do not need to use a transaction manager when communicating with a single database (single or multi-instance). The database does all the work of coordinating the transaction for you.

Managing Transactions with Sessionless Transactions

A Sessionless transaction breaks the coupling between the transaction and the session. Once you start a transaction, it does not need to be tied to a session or connection. You can free the session or connection, allowing it to be used by another client. Additionally, you do not need a transaction manager to coordinate the XA protocol because the 2PC process and the error recovery process are handled automatically by the database. Therefore, there is no risk of in-doubt transactions and no need for any recovery mechanism.

To sum up, the key differentiators of Sessionless Transactions are:

- Efficient use of sessions and connections because a transaction is not sticky on a session or connection.
- Reduced commit latency because the transaction finalization happens internally and fewer client-server round trips are needed.
- Improved transactional throughput because the recovery mechanism is handled internally, thereby eliminating the possibility of locks holding resources indefinitely.
- Transactions never go in-doubt.

31.2 Sessionless Transactions Overview

Sessionless Transactions is a feature for managing transactions efficiently in a database application.

The Sessionless Transactions feature enables users to start a transaction on a database session by providing a unique transaction identifier, submit a unit of work, suspend the transaction, and continue the same transaction on another session using the same transaction identifier. In the end, the same transaction can be committed from yet another session.

The Sessionless Transactions feature provides applications with a native mechanism to commit or roll back a transaction without employing an external transaction manager to coordinate the XA protocol. The transaction commit is coordinated internally by the database to ensure data integrity across multiple database instances or sessions.

You can use Sessionless Transactions on a RAC or a non-RAC deployment. If it is a RAC deployment, each unit of work could use sessions on different database instances. On a non-RAC deployment, you have a single instance, and each unit of work could use different sessions on the same database instance.

The following is an example illustrating the workflow on a RAC deployment.

1. Acquire a session on Instance 1.
2. Set a unique identifier for the transaction.
3. Start a new Sessionless transaction.
4. Submit the unit of work – DML or SQL statements.
5. Suspend the Sessionless transaction.
6. Release the session.
7. Acquire a session on Instance 2.
8. Set the same unique identifier for the transaction.

Copyright © 1999, 2025, Oracle and/or its affiliates.

9. Resume the Sessionless transaction.
10. Submit the unit of work – DML or SQL statements.
11. Suspend the Sessionless transaction.
12. Release the session.
13. Acquire a session on Instance 3.
14. Set the same unique identifier for the transaction.
15. Resume the Sessionless transaction.
16. `COMMIT`.
17. Release the session.

> **See Also**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information about Oracle RAC

To start, suspend, and resume a Sessionless transaction, you can use server-side (PL/SQL) or client-side APIs. The APIs enable you to:

- Set a user-provided or auto-generated unique transaction identifier on a transaction.
- Start a new transaction with the given transaction identifier.
- Suspend the transaction identified by the transaction identifier.
- Resume the transaction identified by the transaction identifier.

These APIs are discussed in detail in the subsequent sections of this document.

31.3 Sessionless Transactions Capabilities

Sessionless Transactions provides the following capabilities.

Multiplexing Transactions between Sessions and Connections

Sessionless Transactions provides native mechanisms to start, suspend, and resume a transaction, and in the end, commit or roll back the transaction, as shown in the following Figure 31-1 and Figure 31-2. Sessionless Transactions gives you the flexibility to eliminate the need for the XA protocol and an external transaction manager. Therefore, you can obtain the key advantages of reduced latency and Oracle Database's high availability.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 31-1 Multiplexing Transactions across Sessions on a RAC Deployment

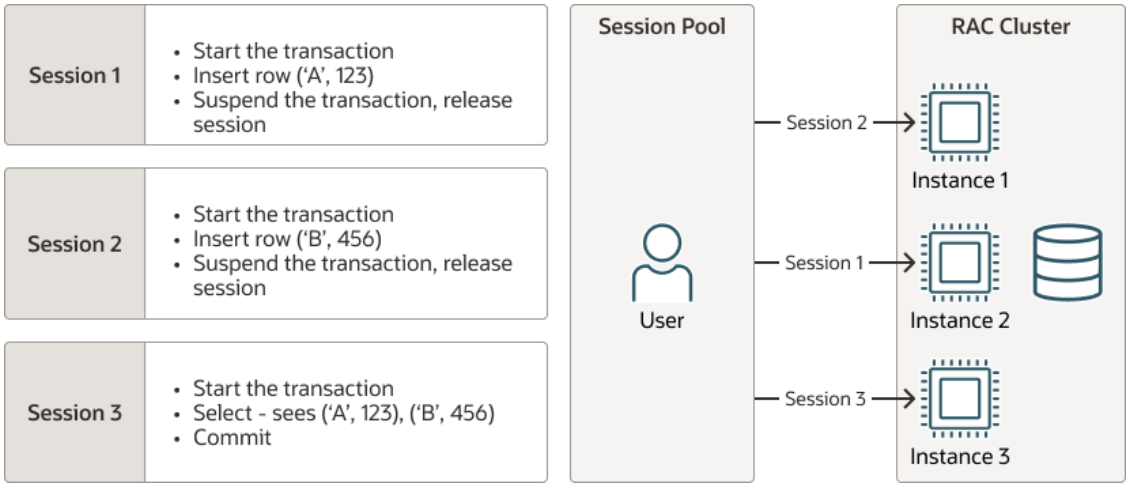
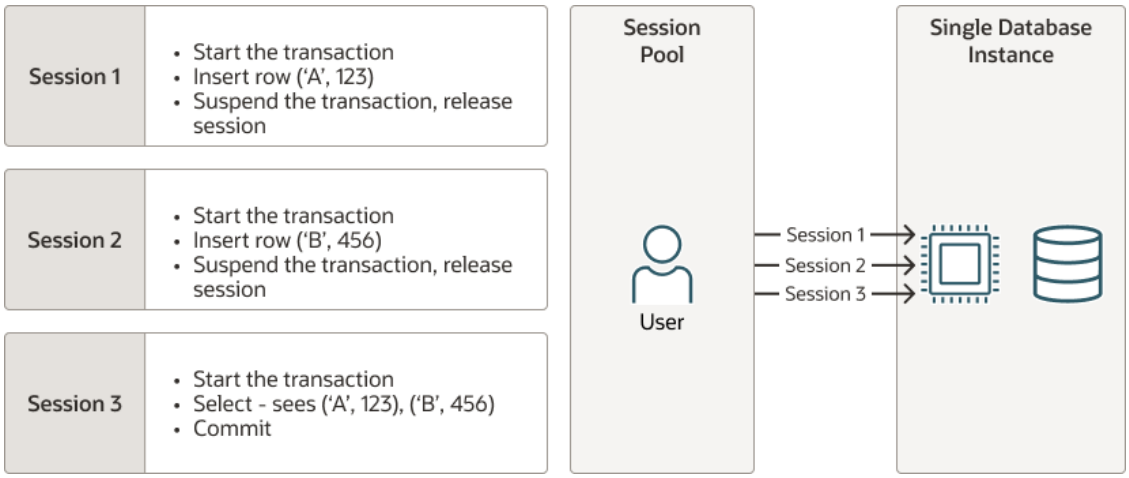


Figure 31-2 Multiplexing Transactions across Sessions on a Single-instance Deployment



Eliminating External Transaction Managers when Using a Single Database

Enterprise applications enlist transaction managers to guarantee completion of a series of operations in transactional resource managers such as Oracle Database. Sessionless Transactions employs 2PC internally to commit or roll back a Sessionless transaction, thereby eliminating the need for an external transaction manager when communicating with the same database. However, a transaction manager is still needed when coordinating a transaction across different databases.

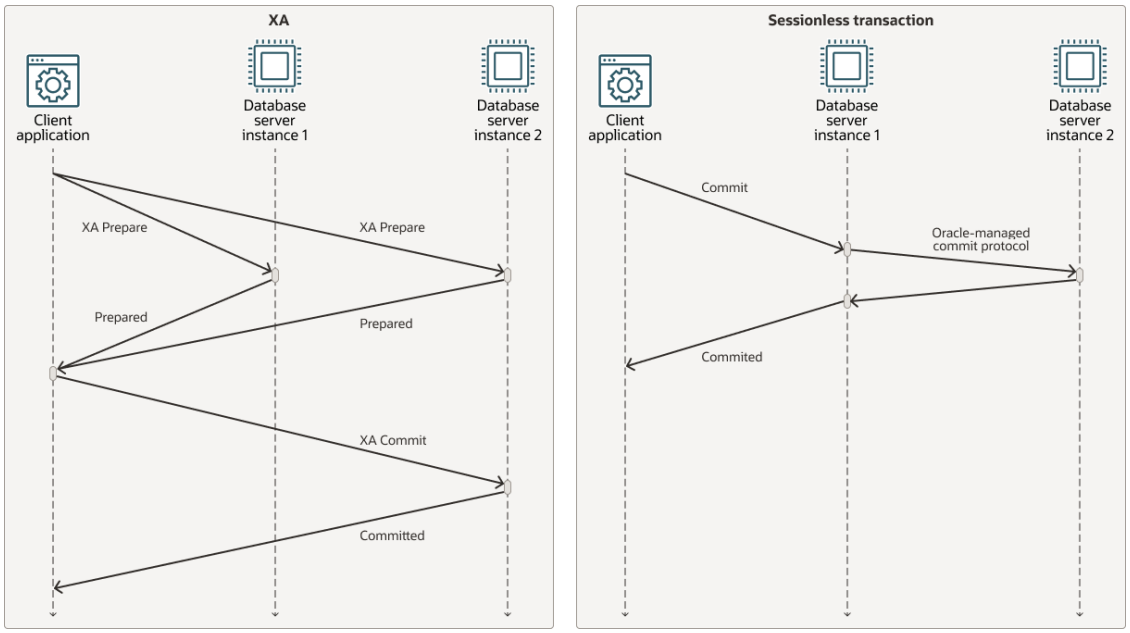
Improving Performance

The Sessionless Transactions feature implements the commit protocol inside Oracle Database, allowing the application or client to issue a simple commit command to finalize a transaction. In fact, the commit command could be issued from any connection or session having the right transaction identifier, thereby emphasizing the feature's connection and session multiplexing capability. The reduced commit latency of Sessionless transactions is in sharp contrast to the XA protocol, where the application or the transaction manager must drive the commit protocol,

Copyright © 1999, 2025, Oracle and/or its affiliates.

making several round trips to the database server, resulting in increased commit latency. As shown in the following Figure 31-3, Oracle internally coordinates 2PC for Sessionless transactions. The clients do not need to issue the prepare round-trip calls, which have higher latencies when compared to communication between Oracle server instances.

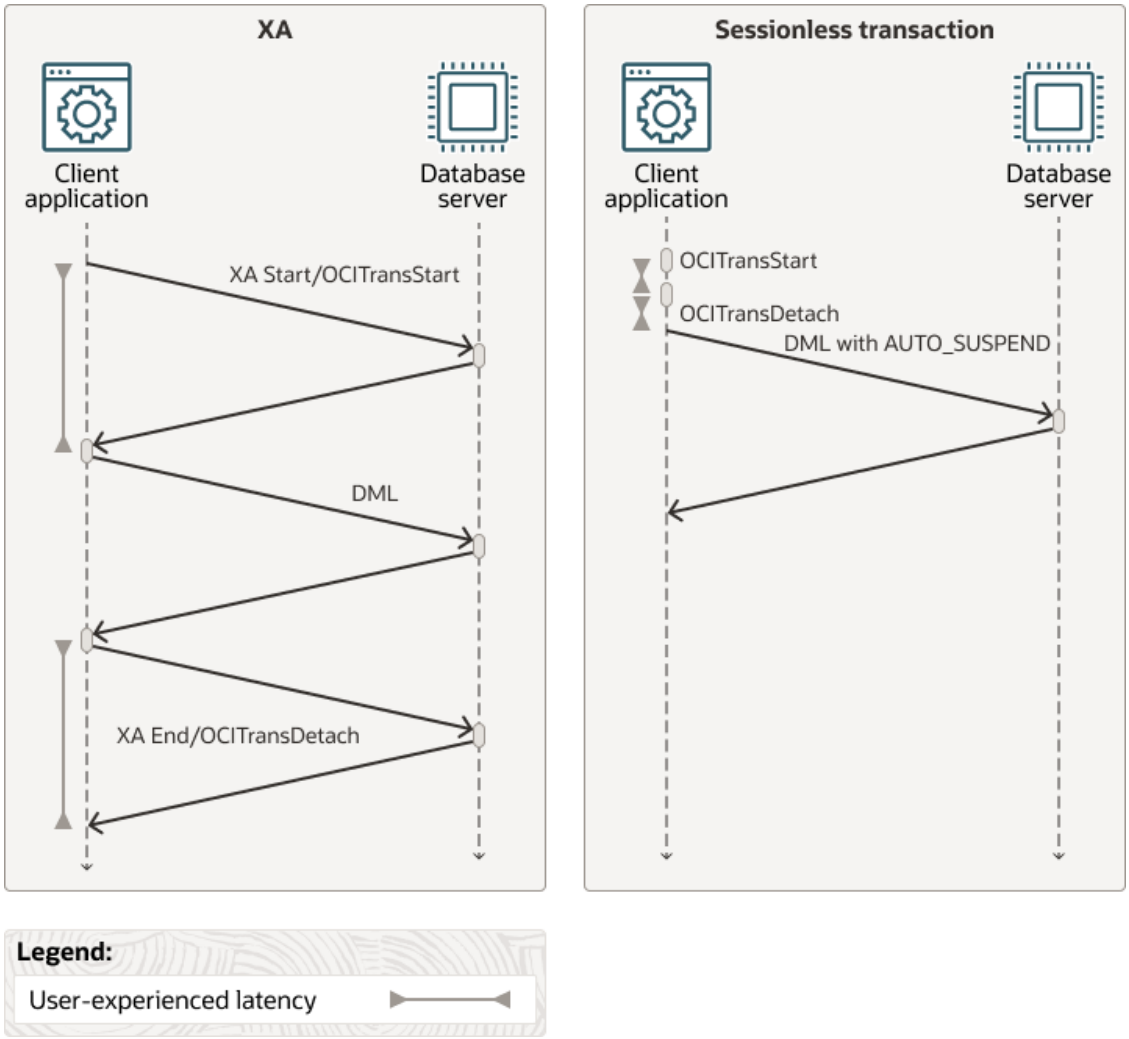
Figure 31-3 XA Transactions and Sessionless Transactions 2PC



In addition to reduced commit latency, the calls to start, resume, or suspend a Sessionless transaction can be attached to the next network round trip. As such, the user-conceived Sessionless transaction's start, resume, and suspend latencies are less than that of the counterparts of XA transactions. Figure 31-4 illustrates the starting and detaching/suspending latency differences between XA transactions and Sessionless transactions. With Sessionless transactions, the `OCITransStart` and `OCITransDetach` calls return immediately without making a round trip to the database server. These requests are sent to the server along with the subsequent server round trip to the database server.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 31-4 XA Transactions and Sessionless Transactions



Building Highly Fault-tolerant and Auto-recoverable Applications

With Sessionless Transactions, the commit protocol is coordinated by the database. Hence, the burden of transaction recovery is removed from the applications (or the transaction managers). The transactions never go in-doubt or into a pending state. Therefore, applications do not need to deploy any recovery mechanisms.

> **See Also**
>
> Managing Distributed Transactions in *Database Administrator's Guide* for information about managing in-doubt transactions

Copyright © 1999, 2025, Oracle and/or its affiliates.

31.3.1 Example Use Cases

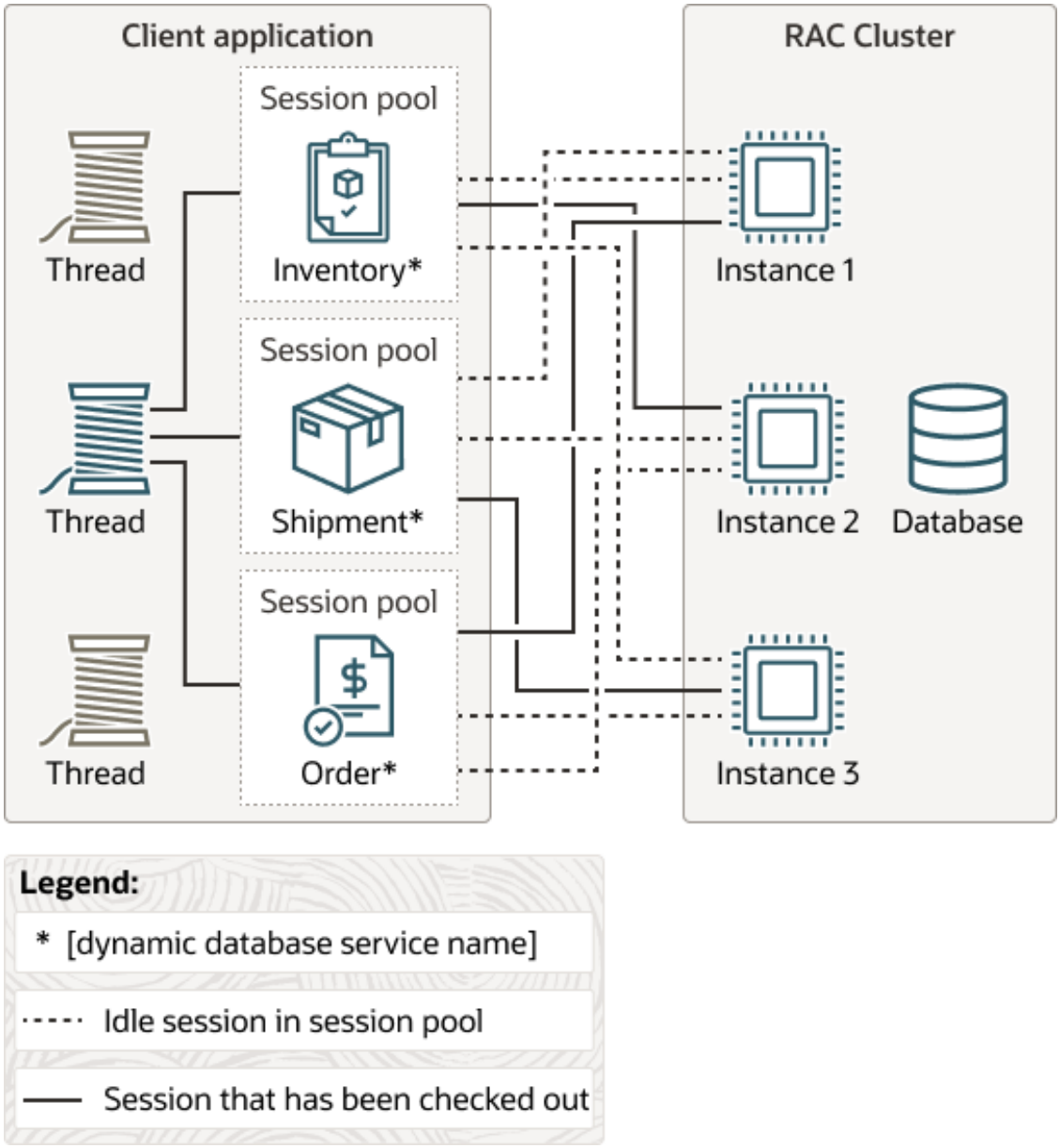
Applications Using Dynamic Database Services

You can create multiple services on a RAC cluster and specify their attributes, such as service-level thresholds and priority. Sessions are connected to a specific service. With the Sessionless Transaction feature, the application can wrap the changes that are done using different services into a single Sessionless transaction. In the following Figure 31-5, a thread in a client application uses multiple sessions that are connected to different services, which make transactional changes.

> **See Also**
>
> Overview of Automatic Workload Management with Dynamic Database Services in RAC Administration and Deployment Guide for more information about Dynamic Database Services

Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 31-5 Dynamic Database Services with Sessionless Transactions



Microservices

Microservices follow the stateless paradigm of application deployment. If several microservices have to be logically combined under a transaction, you have two choices:

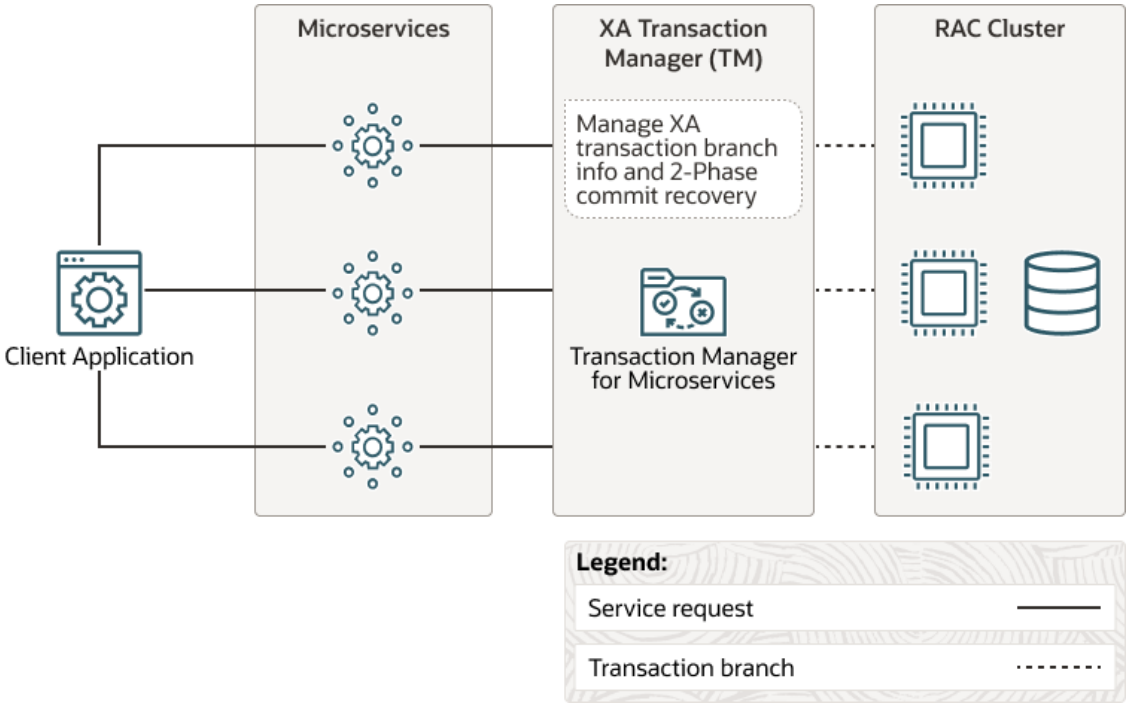
- Deploy external TMs like MicroTx to manage the individual microservices as branches of a global transaction.
- Auto-commit each microservice request. Upon error, an application needs to call compensatory microservices to undo the failed task, which is complex to implement for applications because the recovery logic needs to be handled by the application.

Figure 31-6 and Figure 31-7 respectively demonstrate how to use XA transactions and Sessionless Transactions to combine work from multiple microservices into a single

Copyright © 1999, 2025, Oracle and/or its affiliates.

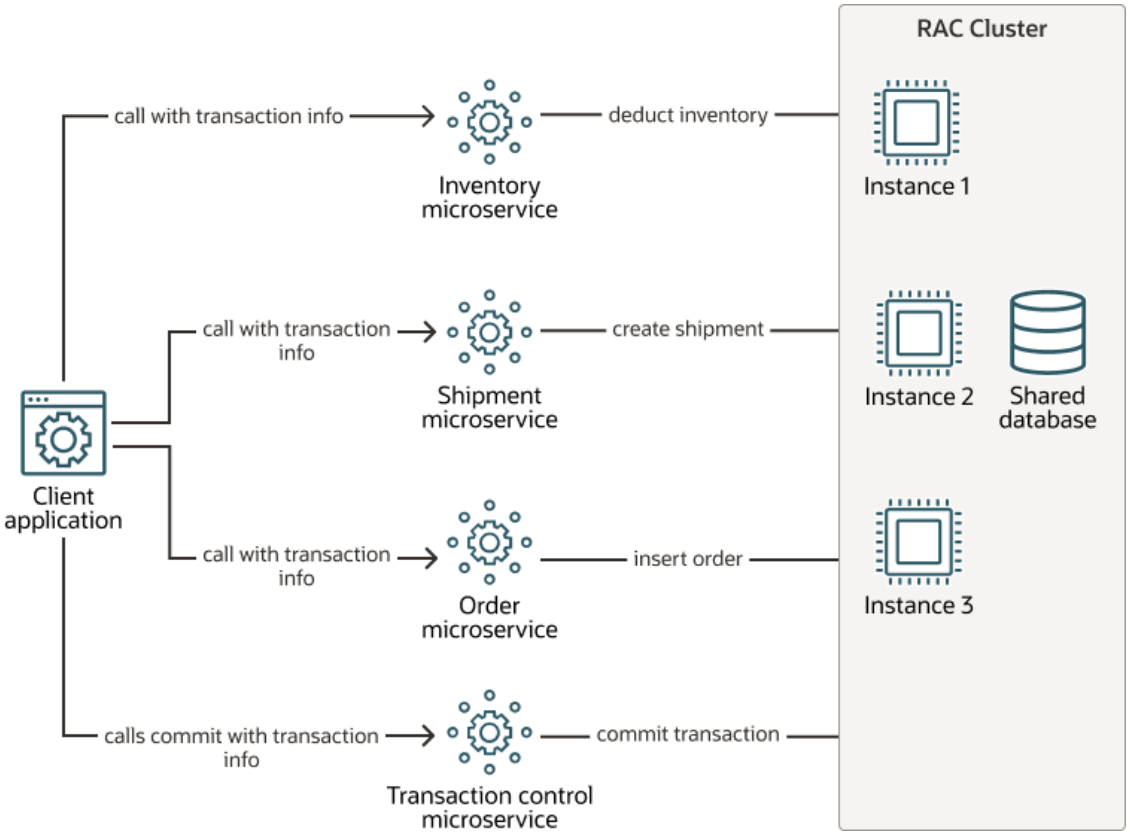
transaction. Sessionless Transactions offers a simpler solution and removes the transaction manager as the middle tier.

Figure 31-6 Microservices with XA Transactions



Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 31-7 Microservices with a Single Sessionless Transaction

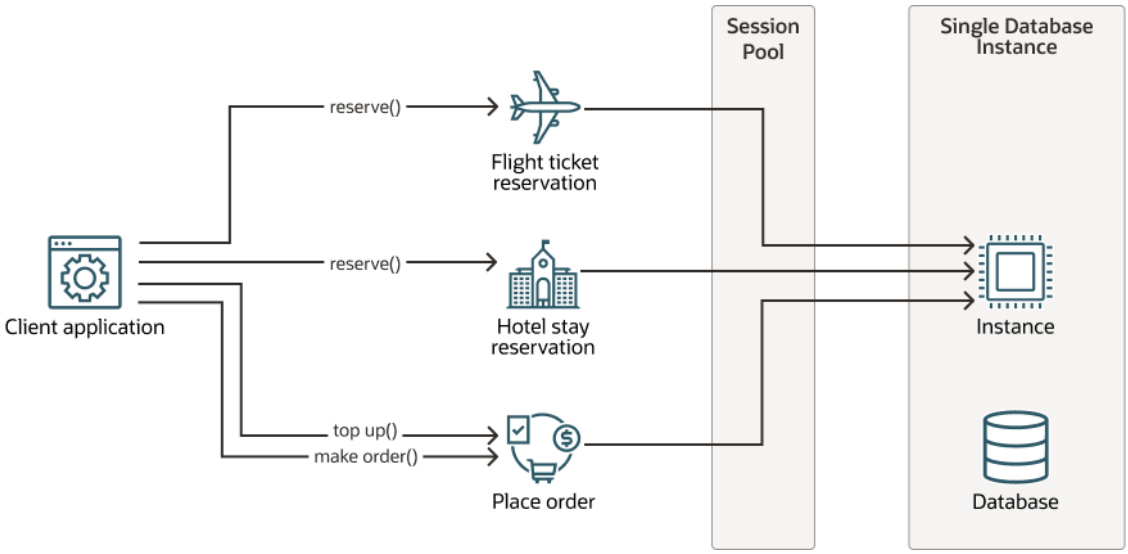


Interactive Applications

An interactive application as an application with think time and computation logic can take advantage of Sessionless Transactions. For example, a travel reservation service may let the customers browse available hotel vacancies and flight tickets. When browsing, the customers may hold the hotel or flight reservations that they are interested in for a certain duration. Once the itinerary is finalized, the customers are asked to pay for the bookings from their account balance. Sessionless transactions use the database sessions only during the actual operation and can release the database sessions during the customer's browsing and input time. As such, the same set of database sessions can serve more concurrent requests. Figure 31-8 and Figure 31-9 depict such use cases on a single-instance deployment and a RAC deployment respectively.

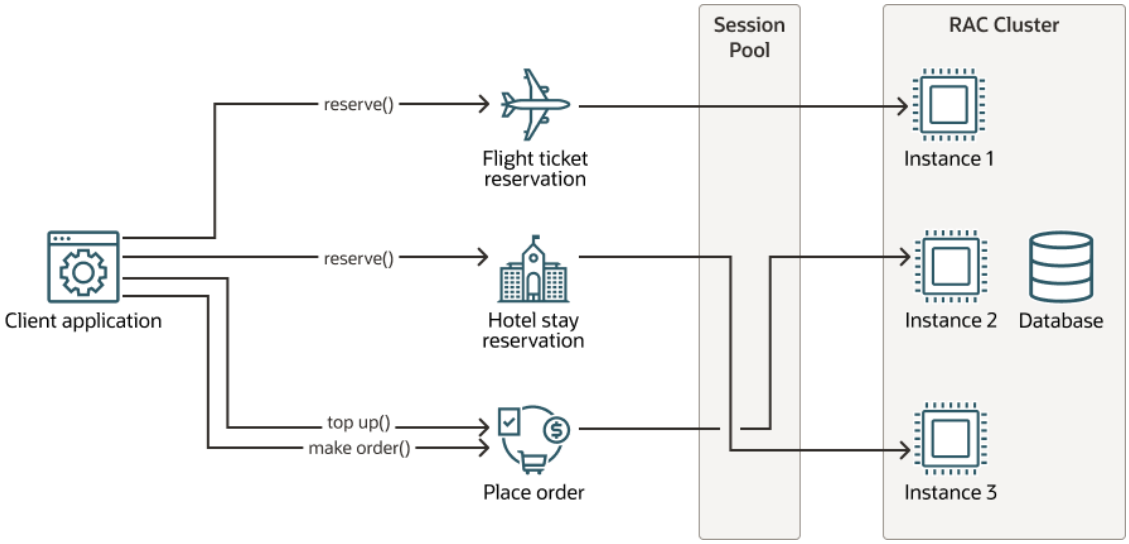
Copyright © 1999, 2025, Oracle and/or its affiliates.

Figure 31-8 Interactive Applications with Sessionless Transactions on a Single-instance Deployment



The interactive application can run on a RAC deployment to enjoy the availability and load balancing provided by RAC.

Figure 31-9 Interactive Applications with Sessionless Transactions on a RAC Deployment



31.4 Benefits of Sessionless Transactions

The following are the benefits of using Sessionless Transactions in your applications.

Efficient Use of Session and Connection Resources

Sessionless Transactions allows applications to multiplex sessions and transactions without the complexity of the eXtended Architecture (XA).

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Developing Applications with Oracle XA for information about Oracle XA

Database-managed Recovery

Oracle Database is responsible for transaction recovery and commit operations for Sessionless transactions. The client application does not require transaction management or an external transaction manager. Transactions also never go in-doubt.

Reduced Network Round Trips between Application Client and (Oracle) Database Server

Using Sessionless Transactions, applications would simply send the commit command and the server manages the commit protocol internally for the transactions. The number of round trips between the client application and the database instances is reduced substantially.

Transaction Can Be Resumed across RAC Database Instances

Unlike the XA protocol, a Sessionless transaction can be started and suspended on one RAC database instance and resumed on another RAC database instance. An application does not need to know about which RAC database instance the transaction uses.

In Oracle XA, to suspend and resume an XA transaction on different RAC database instances, the external transaction managers must have the knowledge of each RAC database instance, create at least one branch per instance, which is used by the XA transaction, and manage the 2PC of these branches. The XA driver cannot suspend an XA transaction on one instance and resume it on another. The users are forced to create branches on a new instance to continue the work under a given XA transaction.

Reduced 2PC Protocol Latency

Sessionless Transactions coordinates commit operations internally, and hence applications can benefit from reduced commit (2PC Protocol) latency because fewer client-server round trips are needed for finalizing transactions.

If an XA transaction is spread across multiple branches, the external transaction manager must send 2PC messages to all those branches.

External Transaction Managers Are Not Required

With Sessionless Transactions, there is no need for an external transaction manager when communicating with one database.

31.5 Using Sessionless Transactions

The following sections explain how to use Sessionless transactions in your applications.

- Understanding Active Sessionless Transactions
- Understanding the Lifecycle of Sessionless Transactions
- Understanding Server Round Trips and Pre-call and Post-call Functions
- Prerequisites for Using Sessionless Transactions
- Setting a Global Transaction ID
- Starting a New Sessionless Transaction

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Retrieving a Global Transaction ID
- Suspending a Sessionless Transaction
- Resuming a Suspended Sessionless Transaction
- Finalizing a Sessionless Transaction
- Example: Sessionless Transactions with OCI API
- Rules and Guidelines for Using Sessionless Transactions
- Error Messages and Notifications

31.5.1 Understanding Active Sessionless Transactions

A Sessionless transaction that is currently associated with a session is said to be an active Sessionless transaction. A Sessionless transaction is active from the time it is newly started or resumed to the time the transaction is suspended, committed, or rolled back.

31.5.2 Understanding the Lifecycle of Sessionless Transactions

You can start a Sessionless transaction on the server (using the PL/SQL functions and procedures in the `DBMS_TRANSACTION` package) or on the client (using client APIs, such as Oracle Call Interface (OCI) or Oracle JDBC). The set of APIs to start, resume, and suspend Sessionless transactions on the server is not interoperable with that on the client during the duration in which the Sessionless transaction is active.

A unique transaction identifier called GTRID (Global Transaction ID, which is also referred to as Transaction ID) identifies each Sessionless transaction. When starting a Sessionless transaction, you can specify the desired GTRID that identifies the transaction. If you do not provide a GTRID, the database server (if you start the Sessionless transaction on the database server) or the client driver (if you start the Sessionless transaction on the client) generates an identifier that is used as the transaction's GTRID. Subsequent units of work submitted to the Oracle Database server are in the transaction context. Over the course of time, the transaction could be suspended and resumed between units of work, multiple times. The transaction ends when the transaction is committed or rolled back.

Unlike local transactions where the row lock waiting period is infinite, in Sessionless Transactions, row lock waiting period is bound by the `DISTRIBUTED_LOCK_TIMEOUT` database parameter. After the timeout, the statement that is waiting on a row lock receives an error, and is rolled back.

31.5.3 Understanding Server Round Trips and Pre-call and Post-call Functions

Server Round Trip

A server round trip is the trip from the client to the server and then back to the client. The user call that incurs the server round trip is the main call. Each server round trip has a main call and may have pre-call functions and post-call functions attached to the main call.

Pre-call and Post-call Functions

For certain user calls, client libraries do not immediately incur a server round trip. Instead, client libraries store the information of the calls and attach the calls on the next server round trip. If the attached call is run before the main call of the server round trip, such a call is calling

Copyright © 1999, 2025, Oracle and/or its affiliates.

a pre-call function, and if the attached call is run after the main call of the server round trip, such a call is calling a post-call function.

Pre-call Functions

An example of a pre-call function is as follows:

```
int main()
{
  ...
  OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);  /*
a pre-call that starts a Sessionless transaction */
  OCIStmtExecute(...) /* a call that incurs a round-trip to server. This is
the "server round trip." */
  ...
}
```

In the preceding example, the first OCI call to start a Sessionless transaction (`OCITransStart`) is a pre-call function. This pre-call returns, after the client library records the call information, without incurring a server round trip. The `OCIStmtExecute` call incurs the server round trip (and therefore is the round trip's main call), on which the `OCITransStart` pre-call function attaches itself. To handle this server round trip, the database server first runs the pre-call function (`OCITransStart`) and then runs the main call (`OCIStmtExecute`).

If any pre-call function raises an error, the main call and the possible post-call functions are not invoked.

Post-call Functions

Post-call functions are attached to the next server round trip and are run after the successful running of the main call.

An example of a post-call function is as follows:

```
int main()
{
   ...
  OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS |
OCI_TRANS_POST_CALL));  /* a post-call which suspends a Sessionless
transaction */
  OCIStmtExecute(...) /* a call that incurs a round-trip to server. This is
the "main call". */
  ...
}
```

In the preceding example, the first OCI call to suspend the Sessionless transaction (`OCITransDetach`) is a post-call function. It is returned immediately after the client library records this call, but the call is not sent to the server yet. `OCIStmtExecute` incurs the server round trip (and therefore is the round trip's main call), on which the post-call function `OCITransDetach` attaches itself. To handle this server round trip, the database server first runs the main call (`OCIStmtExecute`) and then runs the post-call function (`OCITransDetach` with `OCI_TRANS_POST_CALL`).

If the main call raises an error, the post-call functions are not invoked.

Copyright © 1999, 2025, Oracle and/or its affiliates.

31.5.4 Prerequisites for Using Sessionless Transactions

Prerequisites for Using Sessionless Transactions on the Server

There are no prerequisites for using Sessionless Transactions on the server.

Prerequisites for Using Sessionless Transactions on the OCI Client

To use the start, suspend, and resume Sessionless Transactions APIs on the OCI client, you must allocate a transaction handle and set it as the `OCI_ATTR_TRANS` attribute of the service context handle.

```
OCISvcCtx *svchp = NULL;

/* get a service context handle from the session pool. */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0,
              NULL, NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

/* create a transaction handle */
OCITrans    *txnhp  = (OCITrans *)0; /* Sessionless transaction */
OCIHandleAlloc(envhp, (void **)&txnhp, OCI_HTYPE_TRANS, 0, 0);

/* set txn handle as service context handle's OCI_ATTR_TRANS attribute */
OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);
```

31.5.5 Setting a Global Transaction ID

Every Sessionless transaction is identified by a unique transaction identifier called Global Transaction ID (GTRID). You can either provide your desired GTRID or let Oracle generate a GTRID to identify the Sessionless transaction to be started. In either case, you need to set the GTRID field properly so that the client driver or the database server can perform the right action.

Sessionless APIs are provided both on the client side and on the server side, enabling you to maximize the flexibility to independently upgrade to the database server and client versions.

> **Note**
>
> For auto-generated GTRIDs, the client library generates the GTRID for a Sessionless transaction that is started on the client and the database server generates the GTRID for a Sessionless transaction that is started on the database server.

Setting a GTRID on the Database Server

A GTRID is set on the server as a part of starting a Sessionless transaction on the server.

> **See Also**
>
> Starting a New Sessionless Transaction

Copyright © 1999, 2025, Oracle and/or its affiliates.

Setting GTRID on the OCI Client

> **Note**
>
> This should be done before starting the transaction.

If you prefer to identify the new Sessionless transaction with a specific GTRID, create an XID structure, set the GTRID in the XID's `data` member, and set the XID's `gtrid_length` member to the size of the GTRID (in bytes). Then, call `OCIAttrSet` to set transaction handle's `OCI_ATTR_XID` attribute to the pointer of the XID structure, as shown in the following code sample. The client library records the GTRID and stores it in the transaction handle.

Example 31-1 Setting a Specific GTRID

```
/* Get current transaction handle */
OCITrans *current_txnhp;
OCIAttrGet(svchp, OCI_HTYPE_SVCCTX, &current_txnhp, NULL,
           errhp);

/* set GTRID in XID struct */
XID         res_xid;
strcpy(res_xid.data, "client_user_gtrid_1");
res_xid.gtrid_length = strlen("client_user_gtrid_1");

/* set XID as txn handle's attribute */
OCIAttrSet(current_txnhp, OCI_HTYPE_TRANS, &res_xid, sizeof(XID),
OCI_ATTR_XID, errhp);
```

If you prefer to use a generated UUID as the GTRID identifying the new Sessionless transaction, set the transaction handle's `OCI_ATTR_XID` attribute to a `NULL` pointer, as shown in the following code sample. When starting the transaction, a UUID is generated by the client driver and used as GTRID.

If a generated UUID is used as the GTRID, before the transaction gets suspended, you must retrieve the XID from the transaction handle so that you can use it again to resume or commit the transaction.

> **See Also**
>
> Retrieving a Global Transaction ID

Example 31-2 Setting a Generated GTRID

```
/* Get current transaction handle */
OCITrans *current_txnhp;
OCIAttrGet(svchp, OCI_HTYPE_SVCCTX, &current_txnhp, NULL,
           errhp);

/* set XID as txn handle's attribute */
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
OCIAttrSet(current_txnhp, OCI_HTYPE_TRANS, NULL, sizeof(XID), OCI_ATTR_XID,
errhp);
```

You cannot set an XID (`OCI_ATTR_XID`) when:

- There is an active Sessionless transaction that was started or resumed on the database server. In this case, `OCI_ATTR_TRANS` attribute of the service context handle is a read-only transaction handle managed by the OCI library (returns an error 26210).
- An active Sessionless transaction was started with client library generated GTRID, and such GTRID has not been retrieved yet (returns an error 26204).
- A pre-call function to start a new Sessionless transaction or resume an existing Sessionless transaction is recorded but is not yet sent to the database server (returns an error 26216).

> **See Also**
>
> OCIAttrSet() and OCIAttrGet() in *Oracle Call Interface Developer's Guide*

31.5.6 Starting a New Sessionless Transaction

You can start a Sessionless transaction on the server or on the client. If an active Sessionless transaction exists when the request for starting a new transaction is made, such an active transaction is suspended, regardless of whether the new transaction request succeeds or not.

When you start a new Sessionless transaction, you can specify a unique GTRID (or use a generated GTRID) and specify a time-out value to use if the transaction is suspended.

> **See Also**
>
> Time-out of a Suspended Sessionless Transaction for information about time-out of suspended Sessionless transactions

Starting a New Sessionless Transaction on the Database Server

To start a new Sessionless transaction on the server, run the `DBMS_TRANSACTION.START_TRANSACTION` PL/SQL function. Set the flag to `TRANSACTION_NEW` to specify that the request must start a new transaction.

You can start a Sessionless transaction on the server using a user-specified GTRID or a generated GTRID.

To start a Sessionless transaction on the database server using a generated UUID as GTRID, simply set the RAW-type parameter `XID` of the `DBMS_TRANSACTION.START_TRANSACTION` PL/SQL function to `NULL`. The function invokes the GUID generator to create a 16-byte random identifier, which is used as the GTRID. Alternatively, you can use the existing `SYS_GUID()` API to obtain a globally unique identifier and pass it in as a Sessionless transaction's GTRID.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - `DBMS_TRANSACTION` package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function
> - SYS_GUID in *SQL Language Reference Guide*

> **Note**
>
> Programs that use an older client library, which does not support Sessionless transactions can still use Sessionless transactions if the Sessionless transaction is started on a server that supports Sessionless transactions.

Example 31-3 Starting a New Sessionless Transaction on the Server with a User-Specified GTRID

The following example starts a new Sessionless transaction on the server with a specific GTRID:

```
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( UTL_RAW.CAST_TO_RAW('user_specified_gtrid')
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );
  DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
user_specified_gtrid


PL/SQL procedure successfully completed.
```

The following retrieves the transaction ID of the newly created Sessionless transaction.

```
BEGIN
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN

DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION
_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  END IF;
 END;
 /
```

The output is:

```
user_specified_gtrid

PL/SQL procedure successfully completed.
```

Example 31-4 Starting a New Sessionless Transaction on the Server with a Generated GTRID

The following example starts a new Sessionless transaction on the server using a generated GTRID.

```
-- Start a new Sessionless transaction with a GTRID generated by the
-- START_TRANSACTION function.
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( NULL
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );
  DBMS_OUTPUT.PUT_LINE(gtrid);
END;
/
```

The output is:

```
9AF035CCB2024FA6BF17C8CDBF7B6D6C

PL/SQL procedure successfully completed.
```

The following retrieves the transaction ID of the newly created Sessionless transaction.

```
BEGIN
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN

DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION
_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;
 END;
 /
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The output is:

```
9AF035CCB2024FA6BF17C8CDBF7B6D6C

PL/SQL procedure successfully completed.
```

Client Library Behavior when Sessionless Transaction Starts on the Server Side

For the client library that supports Sessionless transactions, once a Sessionless transaction starts on the server, the server informs the client library about when a Sessionless transaction has started, resumed, suspended, or ended. In OCI's case, when the information is received, the service context handle's `OCI_ATTR_TRANS` attribute is changed to a pointer to a transaction handle managed by the OCI client library. This managed transaction handle is read-only and is available only when the corresponding Sessionless transaction is active. Users can retrieve GTRID from this handle (See Retrieving GTRID on the Database OCI client section). Once the Sessionless transaction is no longer active, the `OCI_ATTR_TRANS` attribute is set back to the value that was set before the Sessionless transaction became active.

Starting a New Sessionless Transaction on the OCI Client

To start a new Sessionless transaction on the client side, run the `OCITransStart` function with the (`OCI_TRANS_SESSIONLESS` | `OCI_TRANS_NEW`) flag and the timeout settings. The `OCI_TRANS_SESSIONLESS` flag distinguishes this request from the requests to start an XA branch. When `OCITransStart` is used with the `OCI_TRANS_SESSIONLESS` flag, it is always a pre-call function.

When you start a new Sessionless transaction, you can specify a unique GTRID (or use a generated GTRID) and specify a time-out value to use if the transaction is suspended.

> **See Also**
>
> Setting a Global Transaction ID

Example 31-5 Starting a New Sessionless Transaction on the OCI Client

```
OCISvcCtx *svchp;
OCIError *errhp;
...
ub4 timeout = 60;

/* start the transaction. This will be attached on the subsequent server
round trip.
    */
OCITransStart(svchp, errhp, timeout, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
```

timeout: how long (in seconds) after the suspension of this Sessionless transaction should the server roll back this transaction. This avoids a transaction from holding on to database resources (such as row locks) indefinitely.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> The timeout value must be a positive number.

> **See Also**
>
> - OCITransStart() in *Oracle Call Interface Developer's Guide*
> - Time-out of a Suspended Sessionless Transaction

31.5.6.1 Running SQL Statements within Sessionless Transactions

After starting a new Sessionless transaction or resuming an existing one, all the SQL statements issued are in the context of the Sessionless transaction.

> **Note**
>
> If a SQL statement causes any lock contention when a Sessionless transaction is active (for example, updating a row whose row lock is held by another transaction), the transaction waits for the lock for up to `distributed_lock_timeout` seconds, and then returns an ORA-2049 error. If an error is returned, the statement is rolled back, but other work in the transaction remains intact.

> **See Also**
>
> - Starting a New Sessionless Transaction
> - Resuming a Suspended Sessionless Transaction

31.5.7 Retrieving a Global Transaction ID

You can retrieve the GTRID of a Sessionless transaction that is currently in the active state.

Retrieving GTRID on the Database Server

To retrieve the GTRID of an active Sessionless transaction on the server, use the `DBMS_TRANSACTION.GET_TRANSACTION_ID` PL/SQL function.

Example 31-6 Retrieving GTRID on the Database Server

The following code samples start a Sessionless transaction and use the `GET_TRANSACTION_ID()` function to retrieve the Sessionless transaction's GTRID.

```
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  ( UTL_RAW.CAST_TO_RAW('user_specified_gtrid')
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );
  DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
user_specified_gtrid

PL/SQL procedure successfully completed.
```

The following retrieves the GTRID.

```
BEGIN
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN

DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION
_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;
END;
/
```

The output is:

```
user_specified_gtrid

PL/SQL procedure successfully completed.
```

> **See Also**
>
> `DBMS_TRANSACTION` package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `GET_TRANSACTION_ID` function

Retrieving a GTRID on the OCI Client

To retrieve a GTRID on the OCI client, call the `OCIAttrGet` function to retrieve `OCI_ATTR_XID` attribute of the transaction handle, which is a pointer to an XID structure. Then, read the XID structure's `gtrid_length` member to learn about the size of the GTRID in bytes and the `data` member to get the actual content of the GTRID.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> - If the GTRID was generated by the client library, users must retrieve the transaction's GTRID before suspending the transaction. Users must have the GTRID to resume and commit the transaction.
> - The pointer obtained from `OCIAttrGet` is read-only. Users should set GTRID using the method specified in Setting GTRID on the OCI Client.

Example 31-7 Retrieving GTRID on the OCI Client

```
OCITrans *current_txnhp;
XID      *current_xidp;
char      gtrid_buffer[65];

/* get transaction handle from service context handle */
OCIAttrGet(svchp, OCI_HTYPE_SVCCTX, &current_txnhp, NULL, OCI_ATTR_TRANS,
errhp);

/* get XID from transaction handle */
OCIAttrGet(current_txnhp, OCI_HTYPE_TRANS, &current_xidp, NULL, OCI_ATTR_XID,
errhp));

/* get GTRID from XID */
memcpy(gtrid_buffer, current_xidp->data, current_xidp->gtrid_length);
```

> **See Also**
>
> OCIAttrGet() in *Oracle Call Interface Developer's Guide*

31.5.8 Suspending a Sessionless Transaction

You can suspend an active Sessionless transaction using the same method (on the database server or on the client) that the Sessionless transaction is last started or resumed. The suspend function or procedure can only be applied to Sessionless transactions. If no transaction is in the session, the suspend is a no-op; if a non-sessionless transaction (for example, a local transaction or an XA transaction) is in the session, calling the suspend function raises an ORA-26202 error, but leaves the transaction in the session intact.

Suspending an Active Sessionless Transaction on the Database Server

Run the `DBMS_TRANSACTION.SUSPEND_TRANSACTION` PL/SQL procedure to suspend a Sessionless transaction that is active in a particular session. This procedure does not have any parameters.

> **See Also**
>
> DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `SUSPEND_TRANSACTION` procedure

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 31-8 Suspending an Active Sessionless Transaction on the Server

The following is an example of suspending an active Sessionless transaction on the server:

```
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  -- start Sessionless transaction
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( UTL_RAW.CAST_TO_RAW('user_specified_gtrid')
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );

  -- confirm the transaction is started
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN
    DBMS_OUTPUT.PUT_LINE('After start: ' ||
UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;

  -- suspend the transaction
  DBMS_TRANSACTION.SUSPEND_TRANSACTION;
  -- confirm the transaction is suspended
  DBMS_OUTPUT.PUT_LINE('After suspend: ' ||
DBMS_TRANSACTION.GET_TRANSACTION_ID());
END;
/
```

The output is:

```
After start: user_specified_gtrid
After suspend:

PL/SQL procedure successfully completed.
```

Suspending an Active Sessionless Transaction on the OCI Client

To suspend an active Sessionless transaction on the OCI client, call `OCITransDetach` with an appropriate flag depending on how you would like to suspend the transaction. You can specify flag bits to indicate your required option. When the flag is set to (`OCI_TRANS_SESSIONLESS | OCI_DEFAULT`), the suspend is run immediately. When the flag is set to (`OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL`), the suspend a pre-call function (attached to the next server round trip and run before the main call). When the flag is set to (`OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL`), the suspend is a post-call function (attached to the next server round trip and run after the main call).

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> If the pre-call suspend (`OCI_TRANS_PRE_CALL` is specified) raises an error, the main call is not run. If the main call raises an error, the post-call suspend (`OCI_TRANS_POST_CALL` is specified) is not run.

Example 31-9 Immediate Suspend

To suspend the active Sessionless transaction immediately, call the `OCITransDetach` function with the flag (`OCI_TRANS_SESSIONLESS | OCI_DEFAULT`), as in the following example.

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_DEFAULT);
```

Example 31-10 Pre-call Suspend (Suspend before the Next Server Round Trip)

To issue a pre-call suspend (suspend on the next server round trip, before the main call), call the `OCITransDetach` function with the `OCI_TRANS_SESSIONLESS` | `OCI_TRANS_PRE_CALL` flag. Being a pre-call function, the suspend code is run *before* the main call.

In the following example, the transaction is actually suspended before the `OCIPing()` call.

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL);
OCIPing(svchp, errhp, OCI_DEFAULT);
```

The run sequence of immediate suspend and pre-call suspend are the same, but pre-call suspend is attached to the next server round trip, thus reducing network latency. For example, if you would make a query after suspending a Sessionless transaction, using pre-call suspend can save a server round trip and reduce latency.

Example 31-11 Post-call Suspend (Suspend after the Next Server Round Trip)

To issue a post-call suspend (suspend on the next server round trip, after the main call), call the `OCITransDetach` function with the `OCI_TRANS_SESSIONLESS` | `OCI_TRANS_POST_CALL` flag.

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL);
OCIPing(svchp, errhp, OCI_DEFAULT);
```

Doing this is useful if you know about the last round trip to the server (for example, the last DML of this unit of work).

Example 31-12 Post-call Suspend

In the following example, the transaction identified by the "my_Sessionless_txn3" GTRID is suspended automatically after the successful running of `OCIStmtExecute()`.

```
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0,
               NULL, NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);
OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

memcpy(xidp->data, "my_Sessionless_txn3", strlen("my_Sessionless_txn3"));
xidp->gtrid_length = strlen("my_Sessionless_txn3");
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
// Ask a post-call suspend
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_POST_CALL);

// Do the DML
OCIStmtPrepare(…); /* insert into mytab1(c1, c2) values (1, 1); */
OCIStmtBindByPos(…);
OCIStmtExecute(…);   /* Server suspends the transaction after this DML is
executed. */
```

See Also

- OCITransDetach() and other OCI library functions in *Oracle Call Interface Developer's Guide*
- Resuming a Suspended Sessionless Transaction

31.5.8.1 Time-out of a Suspended Sessionless Transaction

The time-out value of a Sessionless transaction is set when starting a new Sessionless transaction.

Every time the Sessionless transaction is suspended, the timer starts to tick. If a Sessionless transaction is resumed within the time-out period, the timer is reset. If a Sessionless transaction is not resumed within the time-out period, it is rolled back. This is because every transaction in Oracle Database potentially holds resource locks (such as row locks). These resources are held for the duration of the transaction, so if the transaction remains active indefinitely, it will cause outages.

Note

A zero time-out value is not allowed. You must always specify a positive time-out value.

Example 31-13 Timing Out a Sessionless Transaction

This is an example of a Sessionless transaction that has timed out.

The following starts a Sessionless transaction.

```
SET ECHO ON
SET SERVEROUTPUT ON

DROP TABLE IF EXISTS mytab1;
CREATE TABLE mytab1(c1 NUMBER, c2 NUMBER);

-- start a Sessionless transaction with timeout=20 secs
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table.

```
INSERT INTO mytab1(c1, c2) VALUES (1, 1);
```

The output is:

```
1 row created.
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
```

The output is:

```
PL/SQL procedure successfully completed.
```

Trying to resume the transaction fails with error 26218 because the transaction that started has been rolled back due to the timeout.

```
-- idle for 25 seconds
!sleep 25

DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );
END;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The output is:

```
ERROR at line 1:
ORA-26218: sessionless transaction with GTRID
6D795F73657373696F6E6C6573735F74696D656F75745F6578 does not exist.
```

Trying to start a new transaction with the same GTRID succeeds because the previous transaction was rolled back.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

31.5.9 Resuming a Suspended Sessionless Transaction

You can resume a Sessionless transaction that is in a suspended state.

Resuming a suspended Sessionless transaction is similar to starting a Sessionless transaction. The differences between the two are:

- When resuming a Sessionless transaction on the server, in the `DBMS_TRANSACTION.START_TRANSACTION` function, the `TRANSACTION_RESUME` flag replaces the `TRANSACTION_NEW` flag. Similarly, when resuming a Sessionless transaction on the client library, in the `OCITransStart` function, the `OCI_TRANS_RESUME` flag replaces the `OCI_TRANS_NEW` flag.
- The timeout parameter used in resuming a transaction refers to the resume timeout as opposed to the transaction timeout, which is specified during the start of a new Sessionless transaction.

> **See Also**
>
> DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* and Resuming a Sessionless Transaction on the OCI Client for more information about the timeout parameter as used in the `START_TRANSACTION` function and the `OCITransStart` function respectively

If an active Sessionless transaction exists when the resume request is made, such active transaction is suspended before the resume request is run, regardless of whether the resume request succeeds or not.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Resuming a Sessionless Transaction on the Database Server

To resume a suspended Sessionless transaction on the server, run the `DBMS_TRANSACTION.START_TRANSACTION` PL/SQL function. Set the flag to `TRANSACTION_RESUME`.

> **See Also**
>
> DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function

Example 31-14 Resuming a Sessionless Transaction on the Database Server

```
CREATE PROCEDURE PRINT_GTRID IS
BEGIN
    IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN
      DBMS_OUTPUT.PUT_LINE('Sessionless transaction in session: ' ||
UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION_ID()));
    ELSE
      DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
    END IF;
END PRINT_GTRID;
/
```

The output is:

```
Procedure created.
```

The following starts a Sessionless transaction and prints the GTRID.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
    PRINT_GTRID;
    DBMS_OUTPUT.PUT_LINE('gtrid: ' || gtrid);
    DBMS_OUTPUT.PUT_LINE('cast gtrid: ' || UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
Sessionless transaction in session: my_sessionless_timeout_ex
gtrid: 6D795F73657373696F6E6C6573735F74696D656F75745F6578
cast gtrid: my_sessionless_timeout_ex
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table and fetches the updates.

```
INSERT INTO mytab1(c1, c2) values (1, 1);
SELECT * FROM mytab1;
```

The output is:

```
1 row created.

        C1 C2
---------- ----------
         1 1
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
SELECT * FROM mytab1;
```

The output is:

```
PL/SQL procedure successfully completed.

no rows selected
```

The following resumes the transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );

    PRINT_GTRID;
    DBMS_OUTPUT.PUT_LINE('gtrid: ' || gtrid);
    DBMS_OUTPUT.PUT_LINE('cast gtrid: ' || UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
Sessionless transaction in session: my_sessionless_timeout_ex
gtrid: 6D795F73657373696F6E6C6573735F74696D656F75745F6578
cast gtrid: my_sessionless_timeout_ex

PL/SQL procedure successfully completed.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The following fetches the updates made to the table.

```
SELECT * FROM mytab1;
```

The output is:

```
        C1 C2
---------- ----------
         1 1
```

The following rolls back the transaction.

```
ROLLBACK;
```

The output is:

```
Rollback complete.
```

The following drops the `PRINT_GTRID` procedure.

```
DROP PROCEDURE PRINT_GTRID;
```

The output is:

```
Procedure dropped.
```

Resuming a Sessionless Transaction on the OCI Client

Before resuming a Sessionless transaction on the client, the service context handle must have a transaction handle as attribute, and the `OCI_ATTR_XID` attribute must be set in the transaction handle. Calling the `OCITransStart` function with the `OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME` flag and timeout resumes an existing Sessionless transaction. Similar to starting a new Sessionless transaction on the OCI client, the resume call is a pre-call function.

Example 31-15 Resuming a Sessionless Transaction on the OCI Client

```
ub4 timeout = 20;
OCITransStart(svchp, errhp, timeout, OCI_TRANS_SESSIONLESS |
OCI_TRANS_RESUME);
```

timeout: The timeout here refers to the resume timeout, which is different from the transaction timeout. The resume time-out value can be a 0 or a positive number. Within an instance, a Sessionless transaction can only be associated with one session at any given time. Thus, a Sessionless transaction cannot be resumed if it is associated with another session on the same instance. When this situation arises, the resume timeout value specifies how long (in seconds) this session waits for the other session to suspend the transaction, so that it can resume the transaction. If the other session does not suspend the transaction before the wait times out, an ORA-25351 error is raised. If the resume timeout value is 0, the error is immediately raised.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - OCITransStart() in *Oracle Call Interface Developer's Guide*
> - Time-out of a Suspended Sessionless Transaction

31.5.10 Finalizing a Sessionless Transaction

This section explains how you can commit or roll back a Sessionless transaction.

Topics:

- Committing a Sessionless Transaction
- Rolling Back a Sessionless Transaction

31.5.10.1 Committing a Sessionless Transaction

A Sessionless transaction can be ended by all the usual means that end a local transaction. In terms of committing a Sessionless transaction, the SQL statement `COMMIT`, `OCITransCommit`, and `OCIStmtExecute` in `OCI_COMMIT_ON_SUCCESS` mode can be used. Before committing a Sessionless transaction, ensure that no other session is associated with such transaction.

> **See Also**
>
> Commit or Roll Back Operations and OCIStmtExecute() in *Oracle Call Interface Developer's Guide* for information about committing transactions and `OCI_COMMIT_ON_SUCCESS`.

Example 31-16 Committing an Active Sessionless Transaction

In the following example, an anonymous PL/SQL block starts a Sessionless transaction identified by the "my_sessionless_commit_ex_1" GTRID, issues a DML statement, and then commits the transaction.

The following starts a Sessionless transaction.

```
DROP TABLE IF EXISTS mytab1;
CREATE TABLE mytab1(c1 NUMBER, c2 NUMBER);

-- start a Sessionless transaction and print the GTRID
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_commit_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The output is:

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table.

```
INSERT INTO mytab1(c1, c2) values (1, 1);
SELECT * FROM mytab1;
```

The output is:

```
1 row created.

        C1 C2
---------- ----------
         1 1
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
SELECT * FROM mytab1;
```

The output is:

```
PL/SQL procedure successfully completed.
no rows selected
```

The following resumes the transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_commit_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following commits the transaction.

```
COMMIT;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

The output is:

```
Commit complete.
```

Example 31-17 Commit with `OCITransCommit()`

Similarly, to commit an active Sessionless transaction in a session, you can run the `OCITransCommit()` API from the client.

The following example starts a Sessionless transaction identified by the "my_Sessionless_txn5" GTRID, issues a DML statement, and then commits the transaction.

```
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0,
               NULL, NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);
OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

memcpy(xidp->data, "my_Sessionless_txn5", strlen("my_Sessionless_txn5"));
xidp->gtrid_length = strlen("my_Sessionless_txn5");
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);

OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCITransCommit(svchp, errhp, OCI_DEFAULT);
```

Example 31-18 Commit with `OCISessionRelease()`

In keeping with the existing behavior, the `OCISessionRelease` function can also commit an active Sessionless transaction. In the following example, the function is used to commit a Sessionless transaction having the "my_Sessionless_txn6" GTRID.

```
/* start Sessionless transaction */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0, NULL,
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

memcpy(xidp->data, "my_Sessionless_txn6", strlen("my_Sessionless_txn6"));
xidp->gtrid_length = strlen("my_Sessionless_txn6");
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
/* done start Sessionless transaction */

OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCISessionRelease(…); /* Commits the Sessionless transaction */
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 31-19 Commit with `OCI_COMMIT_ON_SUCCESS`

The following example uses `OCI_COMMIT_ON_SUCCESS` to commit a Sessionless transaction.

```
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIBind *bndhp[2];
char *dml = "INSERT INTO mytab1(c1, c2) VALUES (:1, :2)";
int number[2];

...

OCIStmtPrepare2(svchp, &stmthp, errhp, dml, strlen(dml), 0, 0, OCI_NTV_SYNTAX,
                OCI_DEFAULT);
OCIBindByPos(stmthp, &bndhp[0], errhp, 1, &number[0], sizeof(int), SQLT_INT,
0, 0, 0,
             0, 0, OCI_DEFAULT);
OCIBindByPos(stmthp, &bndhp[1], errhp, 2, &number[1], sizeof(int), SQLT_INT,
0, 0, 0,
             0, 0, OCI_DEFAULT);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_COMMIT_ON_SUCCESS);
OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT);
```

See Also

- DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function
- OCITransCommit(), OCISessionRelease(), and OCIStmtExecute() in *Oracle Call Interface Developer's Guide*

31.5.10.2 Rolling Back a Sessionless Transaction

Sessionless transaction can be rolled back by all the usual means that end a local transaction, such as the SQL statements `ROLLBACK`, `OCITransRollback()` and `OCIRequestEnd()`. Before rolling back a Sessionless transaction, ensure that no other session is associated with such transaction.

See Also

OCIRequestEnd() - TAC-23c and OCITransRollback() in *Oracle Call Interface Developer's Guide* for information about `OCIRequestEnd()` and `OCITransRollback()` that enable you to roll back an open transaction

Example 31-20 Rollback Using PL/SQL

In the following example, an anonymous PL/SQL block starts a Sessionless transaction identified by the "my_sessionless_rollback_ex_1" GTRID, issues a DML statement, and rolls back the transaction.

Copyright © 1999, 2025, Oracle and/or its affiliates.

The following code starts the Sessionless transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_rollback_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table and fetches them from the table.

```
INSERT INTO mytab1(c1, c2) values (1, 1);
SELECT * FROM mytab1;
```

The output is:

```
1 row created.

        C1 C2
---------- ----------
         1 1
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
```

The output is:

```
PL/SQL procedure successfully completed.
```

Trying to fetch from the table shows no result.

```
no rows selected
```

Now, let us resume the transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_rollback_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following fetches the transaction details from the table.

```
SELECT * FROM mytab1;
```

The output is:

```
        C1 C2
---------- ----------
         1 1
```

The following rolls back the transaction.

```
ROLLBACK;
```

The output is:

```
Rollback complete.
```

The following verifies that the roll back was indeed successful.

```
SELECT * FROM mytab1;
```

The output is:

```
no rows selected
```

Example 31-21 `OCITransRollback()`

Similarly, to roll back an active Sessionless transaction in a session, run the `OCITransRollback()` API from the client.

The following example starts a Sessionless transaction identified by the "my_Sessionless_txn8" GTRID, issues a DML statement, and then rolls back the transaction.

```
/* start Sessionless transaction */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0, NULL,
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
char gtrid = "my_Sessionless_txn8";
memcpy(xidp->data, gtrid, strlen(gtrid));
xidp->gtrid_length = strlen(gtrid);
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
/* done start Sessionless transaction */

OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCITransRollback(svchp, errhp, OCI_DEFAULT);
```

Example 31-22 Rollback with `OCIRequestEnd()`

In keeping with the existing behavior, the `OCIRequestEnd()` function can also roll back an open Sessionless transaction.

In the following example, the transaction identified by the "my_Sessionless_txn9" GTRID is rolled back after the successful execution of `OCIRequestEnd()`.

```
/* start Sessionless transaction */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0, NULL,
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

char gtrid = "my_Sessionless_txn9";
memcpy(xidp->data, gtrid, strlen(gtrid));
xidp->gtrid_length = strlen(gtrid);
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
/* done start Sessionless transaction */

OCIRequestBegin(…);
OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCIRequestEnd(…); /* Rolled back the Sessionless transaction */
```

See Also

- DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function
- OCITransRollback(), OCIRequestEnd() - TAC-23c, and OCIStmtExecute() in *Oracle Call Interface Developer's Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

31.5.11 Example: Sessionless Transactions with OCI API

The following is an end-to-end example of Sessionless Transactions using the OCI API.

> **Note**
>
> This example requires the `scott.dept` table from the sample schema.

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define USERNAME_LEN 256
#define PASSWORD_LEN 1025
#define DBNAME_LEN   1025

OCIEnv      *envhp  = (OCIEnv *)0;
OCIError    *errhp  = (OCIError *)0;
OCIAuthInfo *authp  = (OCIAuthInfo *)0;
OCISPool    *poolhp = (OCISPool *)0;
OCISvcCtx   *svchp  = (OCISvcCtx *)0;
OCITrans    *txnhp  = (OCITrans *)0;
OCIStmt     *stmthp = (OCIStmt *)0;
OCIBind     *bndhp[3];
OCIDefine   *defhp[3];

char    *username;
char    *password;
char    *dbname;
OraText *pool_name;
ub4      pool_name_len;

void insert_dept(int deptno, const char *dname, const char *loc);
void query_dept(const char *description);
void query_transaction_type(const char *description);
void chk_err(sword ret, OCIError *errhp, sb2 linenum);
#define CHK_ERR(cmd) chk_err(cmd, errhp, __LINE__)

int main(int argc, char **argv)
{
  int nread;
  size_t usernameLen = USERNAME_LEN;
  size_t passwordLen = PASSWORD_LEN;
  size_t dbnameLen = DBNAME_LEN;

  unsigned int   i;
  unsigned char *gtrid;
  unsigned int   gtridlen = 0;
  unsigned char  gtridbuf[64];
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
/* get username, password, dbname (connect string) */
username = malloc(USERNAME_LEN * sizeof(char));
password = malloc(PASSWORD_LEN * sizeof(char));
dbname = malloc(DBNAME_LEN * sizeof(char));
memset(username, 0, USERNAME_LEN);
memset(password, 0, PASSWORD_LEN);
memset(dbname, 0, DBNAME_LEN);

printf("enter username:");
if ((nread = (int)getline((char **)&username, &usernameLen, stdin)) == -1)
{
  printf("ERROR: Failed to get username\n");
  return -1;
}
if(strlen(username) >= 1)
  username[strlen(username)-1] = '\0';
printf("enter password:");
if ((nread = (int)getline((char **)&password, &passwordLen, stdin)) == -1)
{
  printf("ERROR: Failed to get password\n");
  return -1;
}
if(strlen(password) >= 1)
  password[strlen(password)-1] = '\0';
printf("enter dbname:");
if ((nread = (int)getline((char **)&dbname, &dbnameLen, stdin)) == -1)
{
  printf("ERROR: Failed to get dbname\n");
  return -1;
}
if(strlen(dbname) >= 1)
  dbname[strlen(dbname)-1] = '\0';
printf("\n");

/* set up */
{
  int ret;
  /* Create Environment */
  if ((ret = OCIEnvCreate((OCIEnv **)&envhp, (ub4)OCI_DEFAULT | OCI_OBJECT,
                          (dvoid *)0, (dvoid * (*)(dvoid *, size_t))0,
                          (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                          (void (*)(dvoid *, dvoid *))0, (size_t)0,
                          (dvoid **)0)) != OCI_SUCCESS)
  {
    printf("failed to create OCIEnv. ret = %d\n", ret);
    return 1;
  }

  /* Allocate error handle */
  CHK_ERR(OCIHandleAlloc(envhp, (void **)&errhp, OCI_HTYPE_ERROR, 0, 0));

  /* create a session pool and get a session from it */
  CHK_ERR(OCIHandleAlloc(envhp, (void **)&poolhp, OCI_HTYPE_SPOOL, 0, 0));
  CHK_ERR(OCISessionPoolCreate(
    envhp, errhp, poolhp, &pool_name, &pool_name_len, (OraText *)dbname,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      strlen(dbname), 4, 10, 1, (OraText *)username, strlen(username),
      (OraText *)password, strlen((char *)password), OCI_DEFAULT));
    CHK_ERR(OCISessionGet(envhp, errhp, &svchp, NULL, pool_name,
                          pool_name_len, 0, 0, 0, 0, 0, OCI_SESSGET_SPOOL));

    /* allocate a transaction handle and set is as OCI_ATTR_TRANS */
    CHK_ERR(OCIHandleAlloc(envhp, (void **)&txnhp, OCI_HTYPE_TRANS, 0, 0));
    CHK_ERR(
      OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp));
  }

  /* Query dept table before changes. */
  query_dept("before begin");

  /* Start a new Sessionless transaction with a generated GTRID. Insert a row
   * in dept table, then query the table. We should see the inserted row. */
  CHK_ERR(
    OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp));
  CHK_ERR(
    OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW));
  insert_dept(50, "DEVELOPMENT1", "SEATTLE");
  query_dept("in sessionless txn, after insert");

  /* Get the generated GTRID so we can suspend and resume transaction. */
  CHK_ERR(OCIAttrGet(txnhp, OCI_HTYPE_TRANS, &gtrid, &gtridlen,
                     OCI_ATTR_TRANS_NAME, errhp));
  memcpy(gtridbuf, gtrid, gtridlen);
  printf("Generated GTRID in hexadecimal is: ");
  for (i = 0; i < gtridlen; ++i)
    printf("%02X", (unsigned int)gtridbuf[i]);
  printf("\n");

  /* Suspend Sessionless transaction and query dept table. We should not see
   * the inserted row. */
  CHK_ERR(OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_DEFAULT));
  query_dept("outside sessionless txn");

  /* Set OCI_ATTR_TRANS to NULL. */
  CHK_ERR(OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, NULL, 0, OCI_ATTR_TRANS,
errhp));

  /* release the session here and get another session connected to the
   * same database. This sessions can be a different session and can connect
   * to a different instance in RAC configuration. */
  CHK_ERR(OCISessionRelease(svchp, errhp, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCISessionGet(envhp, errhp, &svchp, NULL, pool_name,
                        pool_name_len, 0, 0, 0, 0, 0, OCI_SESSGET_SPOOL));

  /* resume the Sessionless transaction and query dept. We can see the
previous
   * change. */
  CHK_ERR(OCIAttrSet(txnhp, OCI_HTYPE_TRANS, gtrid, gtridlen,
                     OCI_ATTR_TRANS_NAME, errhp));
  CHK_ERR(
    OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp));
  CHK_ERR(
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS |
OCI_TRANS_RESUME));
  query_dept("resumed sessionless txn");

  /* insert an additional row, and commit the Sessionless transaction. */
  insert_dept(51, "DEVELOPMENT2", "SAN FRANCISCO");
  CHK_ERR(OCITransCommit(svchp, errhp, OCI_DEFAULT));

  /* after commit, we can confirm no transaction is active in the session.
when
   * query the table, we can see both rows got inserted. */
  query_transaction_type("after commit");
  query_dept("after commit");

  /* tear down */
  {
    if (txnhp)
      CHK_ERR(OCIHandleFree(txnhp, OCI_HTYPE_TRANS));
    if (svchp)
    {
      CHK_ERR(OCISessionRelease(svchp, errhp, 0, 0, OCI_DEFAULT));
      svchp = NULL;
    }
    CHK_ERR(OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
    if (poolhp)
      OCIHandleFree((void *)poolhp, OCI_HTYPE_SPOOL);
  }

  free(dbname);
  free(password);
  free(username);
  return 0;
}

void insert_dept(int deptno, const char *dname, const char *loc)
{
  OraText dml[] = "INSERT INTO dept(deptno, dname, loc) values (:1, :2, :3)";
  CHK_ERR(OCIStmtPrepare2(svchp, &stmthp, errhp, dml, strlen((char *)dml), 0,
                          0, OCI_NTV_SYNTAX, OCI_DEFAULT));
  CHK_ERR(OCIBindByPos(stmthp, &bndhp[0], errhp, 1, &deptno, sizeof(deptno),
                       SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIBindByPos(stmthp, &bndhp[1], errhp, 2, (void *)dname,
                       strlen(dname) + 1, SQLT_STR, 0, 0, 0, 0, 0,
                       OCI_DEFAULT));
  CHK_ERR(OCIBindByPos(stmthp, &bndhp[1], errhp, 3, (void *)loc,
                       strlen(loc) + 1, SQLT_STR, 0, 0, 0, 0, 0,
OCI_DEFAULT));
  CHK_ERR(OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT));
}

void query_dept(const char *description)
{
  OraText query[] = "SELECT deptno, dname, loc FROM dept ORDER BY deptno";
  int     deptno;
  OraText dname[20];
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  OraText loc[20];
  int     i = 1;
  int     ret;
  printf("DEPT table [%s]\n", description);
  CHK_ERR(OCIStmtPrepare2(svchp, &stmthp, errhp, query, strlen((char *)query),
                          0, 0, OCI_NTV_SYNTAX, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[0], errhp, 1, &deptno, sizeof(deptno),
                         SQLT_INT, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (void *)dname, 20,
                         SQLT_STR, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[2], errhp, 3, (void *)loc, 20,
                         SQLT_STR, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIStmtExecute(svchp, stmthp, errhp, 0, 0, 0, 0, OCI_DEFAULT));
  while ((ret = OCIStmtFetch2((OCIStmt *)stmthp, (OCIError *)errhp, (ub4)1,
                       (ub4)OCI_FETCH_NEXT, (sb4)0, (ub4)OCI_DEFAULT)) == 0)
  {
    printf("Row %d: (%d, %s, %s)\n", i++, deptno, dname, loc);
  }
  if (ret && ret != OCI_NO_DATA)
    CHK_ERR(ret);
  printf("\n");
  CHK_ERR(OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT));
}

void query_transaction_type(const char *description)
{
  OraText query[] =
    "SELECT NVL(TO_CHAR(DBMS_TRANSACTION.GET_TRANSACTION_TYPE), 'NULL') from "
    "dual";
  char txn_type[10];
  int  ret;
  printf("Query transaction type [%s]: ", description);
  CHK_ERR(OCIStmtPrepare2(svchp, &stmthp, errhp, query, strlen((char *)query),
                          0, 0, OCI_NTV_SYNTAX, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[0], errhp, 1, &txn_type,
                         sizeof(txn_type), SQLT_STR, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIStmtExecute(svchp, stmthp, errhp, 0, 0, 0, 0, OCI_DEFAULT));
  while (
    (ret = OCIStmtFetch2((OCIStmt *)stmthp, (OCIError *)errhp, (ub4)1,
                         (ub4)OCI_FETCH_NEXT, (sb4)0, (ub4)OCI_DEFAULT)) == 0)
    printf("%s\n", txn_type);

  if (ret && ret != OCI_NO_DATA)
    CHK_ERR(ret);
  printf("\n");
  CHK_ERR(OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT));
}

void chk_err(sword ret, OCIError *errhp, sb2 linenum)
{
  OraText msgbuf[1024];
  sb4     errcode;
  memset((void *)msgbuf, '\0', 1024);
  if (ret == OCI_ERROR || ret == OCI_SUCCESS_WITH_INFO)
    OCIErrorGet((dvoid *)errhp, (ub4)1, (OraText *)NULL, &errcode, msgbuf,
                (ub4)sizeof(msgbuf), (ub4)OCI_HTYPE_ERROR);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  if (ret == OCI_SUCCESS)
    return;

  printf("-----------------------------------------------------------\n");
  printf(":Line # %d: ", linenum);
  switch (ret)
  {
  case OCI_ERROR:
  case OCI_SUCCESS_WITH_INFO:
    printf(ret == OCI_SUCCESS_WITH_INFO ? "OCI_SUCCESS_WITH_INFO\n"
                                        : "OCI_ERROR\n");
    printf("%s\n", msgbuf);
    break;

  case OCI_NEED_DATA:
    printf("OCI_NEED_DATA\n");
    break;

  case OCI_NO_DATA:
    printf("OCI_NO_DATA\n");
    break;

  case OCI_INVALID_HANDLE:
    printf("OCI_INVALID_HANDLE\n");
    exit(1);

  case OCI_STILL_EXECUTING:
    printf("OCI_STILL_EXECUTING\n");
    break;

  case OCI_CONTINUE:
    printf("OCI_CONTINUE\n");
    break;

  default:
    printf("Error Code %d\n", ret);
    break;
  }
}
```

The output is as follows. Note that your output may have a different username, password, dbname (TNS name), and generated GTRID.

```
enter username:scott
enter password:tiger
enter dbname:cdb1_pdb1

DEPT table [before begin]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
DEPT table [in sessionless txn, after insert]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
Row 5: (50, DEVELOPMENT1, SEATTLE)

Generated GTRID in hexadecimal is: 688330D5C1764F96BFF8267F58E5709E
DEPT table [outside sessionless txn]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)

DEPT table [resumed sessionless txn]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
Row 5: (50, DEVELOPMENT1, SEATTLE)

Query transaction type [after commit]: NULL

DEPT table [after commit]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
Row 5: (50, DEVELOPMENT1, SEATTLE)
Row 6: (51, DEVELOPMENT2, SAN FRANCISCO)
```

31.5.12 Rules and Guidelines for Using Sessionless Transactions

These rules and guidelines apply to the OCI client API.

The Sessionless Transactions feature enforces the following rules and guidelines when using Sessionless transactions with the OCI client API.

Restrictions on Current Transaction Handle

When a Sessionless transaction is active, you are not allowed to change the `OCI_ATTR_TRANS` attribute of service context handle or free the transaction handle.

Rules for OCI Call Functions

The OCI call functions are distinguished as follows:

- Main call - this function runs immediately
- Pre-call - this function runs attached to the next server round trip, before the call
- Post-call - this function runs attached to the next server round trip, after the call

Last Pre-call Function Call Wins

For a single round trip, you can run a maximum of one pre-call function (which can be either start, resume, or suspend), and one post-call suspend. Before a server round trip, if there are

Copyright © 1999, 2025, Oracle and/or its affiliates.

multiple calls to start, resume, or pre-call suspend before the next server round trip is made, only the last call is sent to the database server.

Following is an example where only a pre-call resume [`OCITransStart(OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME)`] and a post-call suspend [`OCITransDetach(OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL)`] calls are sent to the server.

```
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL); /*
this overrides the above OCITransStart call */

memcpy(xidp->data, "GTRID-2", strlen("GTRID-2"));
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME); /*
this overrides the above OCITransDetach(PRE_CALL) call */

OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL); /*
does NOT override anything since it's a post-call */
OCIStmtPrepare2(svchp, &stmthp, errhp, dml, strlen(dml), 0, 0,
OCI_NTV_SYNTAX, OCI_DEFAULT);
OCIBindByPos(…);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_DEFAULT);
                                    /* This is a server round trip that
attaches the pre-call resume and post-call suspend */
```

Start Transaction with Pre-call Suspend is Disallowed if a Post-call Suspend is Recorded

On the client, when post-call suspend is called and has not been attached to the next server round trip, you cannot request to start a new Sessionless transaction, resume an existing Sessionless transaction, or request to pre-call suspend a Sessionless transaction before the next server round trip. Make a server round trip (for example, call `OCIPing()`) before doing so. Following is an example:

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL); /*
a post-call suspend */
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW); /*
this is prohibited and will throw 26215 error */


OCIPing(svchp, errhp, OCI_DEFAULT); /* This incurs a server round trip that
attaches post-call suspend */
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW); /* no
error thrown*/
```

Request Boundaries Behavior

Calls that define request boundaries are `OCIRequestBegin`, `OCIRequestEnd`, and `OCISessionRelease`. Sessionless transactions that are active in a session with respect to these request boundaries currently follow the behavior that is applicable to local transactions in a session.

Copyright © 1999, 2025, Oracle and/or its affiliates.

31.5.13 Error Messages and Notifications

The following table outlines the errors that you might see while using Sessionless transactions, the description of when these errors are displayed, and the next steps you can take to resolve these errors. However, this is not an exhaustive list of all possible errors.

Table 31-1 Errors Summary

Error Codes	Cause	Resolution
ORA-26202	Suspending a transaction that is not a Sessionless transaction (but a local transaction or an XA transaction branch)	Ensure that the transaction being suspended is a Sessionless transaction.
ORA-26204	The active Sessionless transaction was started using a GTRID generated by the client library, and the GTRID has not been retrieved by the user. Actions that suspend the active transaction make users unable to access the transaction forever and are disallowed.	Retrieve the GTRID before doing the action. To learn how to retrieve a GTRID, see the section "Retrieving a GTRID on the OCI client" in the Retrieving a Global Transaction ID section of this document.
ORA-26207	Starting or suspending a Sessionless transaction when the server does not support the Sessionless Transactions feature	Ensure that you are using Oracle Database 23ai, 23.6, and later versions.
ORA-26210	The `OCI_ATTR_TRANS` attribute cannot be altered when a Sessionless transaction that is started or resumed on the database server, is active. In such a case, a read-only transaction handle is created and set as service context handle's `OCI_ATTR_TRANS` attribute. Once this Sessionless transaction is no longer active, the original `OCI_ATTR_TRANS` attribute is placed back and can be altered.	Suspend or end the active Sessionless transaction that was started or resumed on the database server.
ORA-26211	The method (on the database server or on the client) used to start the active Sessionless transaction is different from the method that is used for the current start or suspend call.	Use the same method that started the active Sessionless transaction to suspend the active transaction first.
ORA-26213	When calling start transaction functions, you did not specify (using the appropriate flag) that the call is meant to start a new Sessionless transaction or to resume an existing one.	In the flag, ensure that you specify whether you intend to start a new Sessionless transaction or resume an existing one. For `OCITransStart`, use the right flag: `OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW` or `OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 31-1 (Cont.) Errors Summary

Error Codes	Cause	Resolution
ORA-26214	When suspending the Sessionless transaction on the client, you did not specify whether the call is to be made immediately, attached to the server round trip, and run before the main call (pre-call suspend), or attached to the next server round trip, and run after the main call (post-call suspend).	When suspending a Sessionless transaction, in addition to `OCI_TRANS_SESSIONLESS`, ensure that either `OCI_DEFAULT`, `OCI_TRANS_PRE_CALL`, or `OCI_TRANS_POST_CALL` is specified in the `flag` parameter.
ORA-26215	A suspend is recorded to be run after the next server round trip. Before the suspend is done, starting a new or resuming an existing Sessionless transaction is not allowed. Also, a suspend that runs before the next server round trip is not allowed.	Incur a server round trip.
ORA-26216	A start new or resume call is recorded but not run yet. Therefore, `OCI_ATTR_XID` and `OCI_ATTR_TRANS_NAME` cannot be changed.	Incur a server round trip.
ORA-26217	Attempting to start a new transaction identified by a GTRID, but on the database server a Sessionless transaction identified by such a GTRID already exists	Use a unique GTRID to start a new Sessionless transaction.
ORA-26218	Attempting to resume an existing transaction identified by a GTRID, but on the database server such a Sessionless transaction does not exist	Ensure that the Sessionless transaction identified by the GTRID for the transaction you want to resume, actually exists.
ORA-26219	Attempting to change the `OCI_ATTR_TRANS` attribute of service context handle when a Sessionless transaction is active	`OCI_ATTR_TRANS` can be set only when no Sessionless transaction is active.
ORA-26220	Attempting to free the transaction handle when a Sessionless transaction is active	Free the transaction handle only when no Sessionless transaction is active.

31.6 Sessionless Transactions and Oracle Coordinated Distributed Transaction Interoperability

Sessionless Transactions allows access to remote database objects using database links, either directly or indirectly. An Oracle Coordinated Distributed transaction starts when database links are used to access tables on a remote Oracle Database. If Sessionless transactions access remote database objects, Oracle Database seamlessly manages multi-level (Sessionless and Oracle-coordinated transactions) hierarchical distributed transactions during the 2PC protocol operations.

Copyright © 1999, 2025, Oracle and/or its affiliates.

31.7 Restrictions for Sessionless Transactions

Promoting a Sessionless Transaction to an XA Transaction Is Not Supported

A local transaction can be promoted to an XA transaction. When that happens, the original local transaction becomes a transaction branch of the XA transaction. However, a Sessionless transaction cannot be promoted to an XA transaction in Oracle Database 23ai, Version 23.6. *This restriction will be addressed in the near future.*

Rollback to Savepoint Is Not Supported

Rollback to savepoint is not supported with Sessionless Transactions in Oracle Database 23ai, Version 23.6 under multi-instance RAC configuration. Under such configurations, if a Sessionless transaction savepoint is created in one session and the transaction is later resumed in another session, the savepoint cannot be used. When a rollback to savepoint is called, an ORA-1086 error is raised.

Session States Are Not Carried Across Sessions by Sessionless Transactions

When resuming a Sessionless transaction in a session that is different from the session the transaction was last suspended from, the session states (such as, all parameters set by `ALTER SESSION`, NLS settings, Temp LOB states, PL/SQL states, and any UGA state) are not carried over to the new session. Thus, if you suspend a Sessionless transaction in one session and resume it in another session, you should set up the session states properly. You must ensure that you re-establish the same session states to retrieve the same results (such as NLS settings).

Insert Direct Load, Online Direct Load, and Parallel DML Are Serialized

As of Oracle Database 23ai, Version 23.6, when Insert Direct Load (IDL) or Online Direct Load (ODL) operation is done in a Sessionless transaction, the operation falls back to the traditional insertion method. Likewise, when a Parallel DML (PDML) statement is issued in a Sessionless transaction, such DML runs in a serial manner. However, some of these restrictions will be removed in the upcoming releases.

Database Link Support

To use Sessionless transactions with Database Links (DBLink), either of the following must be true:

- The DBLink must be a shared database link.

  > **See Also**
  >
  > CREATE DATABASE LINK in *SQL Language Reference Guide*

- The source database (where the DBLink is initiated) is running as Shared Server Architecture.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- Shared Server Architecture in *Database Concepts Guide*
- Configuring a Shared Server Architecture in *Net Services Administrator's Guide*

Application Continuity and Transparent Application Continuity Are Not Supported

Application Continuity and Transparent Application Continuity are not supported with Sessionless Transactions. When a Sessionless Transaction is active, Application Continuity pauses recording the operations and does not replay the operations on an instance failure. Application Continuity is enabled again when the Sessionless Transaction is suspended, committed, or rolled back.

You can, however, drain Sessionless transactions during a planned maintenance.

See Also

Draining Sessionless Transactions

31.8 Draining Sessionless Transactions

Application Continuity and Transparent Application Continuity do not support Sessionless transactions, which means that an active Sessionless transaction in a session does not fail over on system failure or service shutdown.

During a planned maintenance:

- You can use draining sessions to resume a suspended Sessionless transaction, enabling you to end (commit or roll back) the transaction before the planned maintenance.
- You cannot use draining sessions to start a new Sessionless transaction, which might not have ended before the planned maintenance.

  If draining sessions are used to start a new Sessionless transaction, an ORA-28 or ORA-3113 error is raised. To start the new Sessionless transaction again, your application must get a new session from the pool, which must be from non-draining instances.

Application Continuity and Transparent Application Continuity support the draining of borrowed (checked-in) session or connection using connection tests. You can enable the use of a connection test for Sessionless transactions and ensure that the borrowed connections allow the suspended Sessionless transactions to resume and disallow new Sessionless transactions.

To enable or disable the use of the connection test for Sessionless transactions, run the `DBMS_APP_CONT_ADMIN.SESSIONLESS_START_TEST` PL/SQL procedure, as follows:

Run the following SQL statement to enable Sessionless transactions draining start test:

```
execute
dbms_app_cont_admin.enable_connection_test(dbms_app_cont_admin.sessionless_sta
rt_test);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Run the following SQL statement to disable Sessionless transactions draining start test:

```
execute
dbms_app_cont_admin.disable_connection_test(dbms_app_cont_admin.sessionless_st
art_test);
```

See Also

- The DBMS_APP_CONT_ADMIN package in *PL/SQL Packages and Types Reference* for more information about the `SESSIONLESS_START_TEST` procedure
- Server Draining Ahead of Planned Maintenance in *Real Application Clusters Administration and Deployment Guide* for more information about draining Sessionless transactions using connection tests

Copyright © 1999, 2025, Oracle and/or its affiliates.

32
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

32.1 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).

32.1.1 Example: Displaying Dependent and Referenced Object Types

Example 32-1 shows how to display the dependent and referenced object types in your database (if you are logged in as DBA).

Example 32-1 Displaying Dependent and Referenced Object Types

Display dependent object types:

```
SELECT DISTINCT TYPE
FROM DBA_DEPENDENCIES
ORDER BY TYPE;
```

Result:

```
TYPE
------------------
DIMENSION
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
EVALUATION CONTXT
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 CONTXT
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.

32.1.2 Example: Schema Object Change that Invalidates Some Dependents

As an example of a schema object change that invalidates some dependents but not others, consider the two views in the following example, which are based on the `HR.EMPLOYEES` table.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 32-2 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 32-2 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       INVALID

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
```

32.1.3 Example: View That Depends on Multiple Objects

A view depends on every object referenced in its query. The view in Example 32-3 depends on the tables employees and departments.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 32-3 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;
```

Note the following:

- `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'
  ```

32.2 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

32.3 Object Status

Every database object has a status value described in Table 32-1.

Table 32-1 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

32.4 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.
- *`object`* is a synonym and the statement does not change the target object.

The operations in the left column of Table 32-2 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 32-2, cause coarse-grained invalidation.

Table 32-2 Operations that Cause Fine-Grained Invalidation

Operation	Exceptions
`ALTER TABLE` *`table`* `ADD` *`column`*	• Dependent object (except a view) uses `SELECT *` on *`table`*. • Dependent object uses *`table`*`%rowtype`. • Dependent object performs `INSERT` on *`table`* without specifying column list. • Dependent object references *`table`* in query that contains a join. • Dependent object references *`table`* in query that references a PL/SQL variable.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 32-2 (Cont.) Operations that Cause Fine-Grained Invalidation

Operation	Exceptions
ALTER TABLE *table* {MODIFY|RENAME|DROP|SET UNUSED} *column* ALTER TABLE *table* DROP CONSTRAINT *not_null_constraint*	• Dependent object directly references *column*. • Dependent object uses SELECT * on *table*. • Dependent object uses *table*%ROWTYPE. • Dependent object performs INSERT on *table* 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 *view* Online Table Redefinition (DBMS_REDEFINITION)	Column lists of new and old definitions differ, and at least one of these is true: • Dependent object references column that is modified or dropped in new view or table definition. • Dependent object uses *view*%rowtype or *table*%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 *synonym*	• New and old *synonym* targets differ, and one is not a table. • Both old and new *synonym* targets are tables, and the tables have different column lists or different privilege grants. • Both old and new *synonym* 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.
DROP INDEX	• 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 *table* 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 32-2 (Cont.) Operations that Cause Fine-Grained Invalidation

Operation	Exceptions
`CREATE OR REPLACE {PROCEDURE|FUNCTION}`	Call signature changes. Call signature is the parameter list (order, names, and types of parameters), return type, `ACCESSIBLE BY` clause ("white list"), purity[1], determinism, parallelism, pipelining, and (if the procedure or function is implemented in C or Java) implementation properties.
`CREATE OR REPLACE PACKAGE`	• `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 number[2], changed. 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. • 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 purity[1] 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 32-2 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`
- Any `UTL_RECOMP` subprogram

Topics:

- Session State and Referenced Packages
- Security Authorization

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database PL/SQL Language Reference* for information about PL/SQL compilation parameter settings
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_UTILITY.COMPILE_SCHEMA`
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `UTL_RECOMP` subprogram

32.4.1 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

32.4.2 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.

32.5 Guidelines for Reducing Invalidation

To reduce invalidation of dependent objects, follow these guidelines:

- Add Items to End of Package
- Reference Each Table Through a View

32.5.1 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;
/
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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;
/
```

32.5.2 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.

32.6 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 32-1.

Topics:

- Revalidation of Objects that Compiled with Errors
- Revalidation of Unauthorized Objects
- Revalidation of Invalid SQL Objects
- Revalidation of Invalid PL/SQL Objects

32.6.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

32.6.2 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.

32.6.3 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.

32.6.4 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.

32.7 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.

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:

   a. 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.

   b. Search for a public synonym that matches the first piece.

      If found, go to step 2. Otherwise, go to step d.

   c. 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

d. 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
> - Name Resolution for Editioned and Noneditioned Objects

32.8 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.

32.9 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Dependencies Among Other Remote Objects
- Dependencies of Applications

32.9.1 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 or signature checking.

The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` determines whether time stamps or signatures govern remote dependencies.

See Also

- *Oracle Database PL/SQL Language Reference*
- Time-Stamp Dependency Mode
- RPC-Signature Dependency Mode

32.9.2 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.

However, the local procedure may be invalidated when a new procedure that references the same remote table is created. The reason being the local procedure is actually dependent on the stub object representing the remote table and it is invalidated when the stub object's timestamp is modified. The stub object's timestamp refresh is strictly limited to scenarios such as create, alter, drop and reload dependencies. Therefore, though execution of the procedure does not refresh the timestamp of the stub object (in order to avoid huge overhead causing performance degrade), it happens during the creation of another local procedure referring the remote table or compilation of the current procedure itself.

32.9.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Manuals for your application development tools and your operating system for more information about managing the remote dependencies within database applications

32.10 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

32.10.1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

32.10.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

32.10.2.1 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

However, if your application requires that callers get the new default value, you must recompile the called procedure.

32.10.2.2 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
```

32.10.2.3 Changing Subprogram Body

Changing the body of a subprogram does not change the RPC signature of the subprogram.

Example 32-4 changes only the body of the procedure `get_hire_date`; therefore, it does not change the RPC signature of `get_hire_date`.

Example 32-4 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;
/
```

32.10.2.4 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 32-3 lists the data type classes and the data types that comprise them. Data types not listed in Table 32-3, such as `NCHAR`, do not belong to a data type class. Changing their type always changes the RPC signature.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 32-3 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
Datetime	DATE TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE INTERVAL YEAR TO MONTH INTERVAL DAY TO SECOND

Example 32-5 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 32-5 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  FROM employees
  WHERE employee_id = emp_number;
END;
/
```

32.10.2.5 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 32-6 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 32-6 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;
/
```

32.10.3 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

32.10.3.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

signature of the called subprogram. If they do not match (using the criteria described in Changing Data Type Classes of Parameters), then an error is returned to the calling session.

32.10.3.2 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.

32.11 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

33
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.

Topics:

- Overview of Edition-Based Redefinition
- Editions
- Editions and Audit Policies
- Editioning Views
- Crossedition Triggers
- Displaying Information About EBR Features
- Using EBR to Upgrade an Application

33.1 Overview of Edition-Based Redefinition

To upgrade an application while it is in use, you must copy the database objects that comprise the database component of the application and redefine the copied objects in isolation. Your changes do not affect users of the application—they can 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 Editioned and Noneditioned Objects), 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**.

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.2 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`.

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

> **See Also**
>
> *Oracle Database Administrator's Guide* for information about CDBs and PDBs

33.2.1 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.

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`.)

Copyright © 1999, 2025, Oracle and/or its affiliates.

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. If the context is source code, then the edition is the one in which the object is actual.

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

See Also

- Enabling Editions for a User
- Current Edition and Session Edition
- Editioned Objects and Copy-on-Change

33.2.1.1 Name Resolution for Editioned and Noneditioned Objects

To try to resolve an object name, Oracle Database uses the procedure described in Name Resolution in Schema Scope. 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 Noneditioned Objects That Can Depend on Editioned Objects), 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

- Current Edition and Session Edition
- Understanding Schema Object Dependency, for general information about dependencies among schema objects, including invalidation, revalidation, and name resolution

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.2.1.2 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

33.2.1.2.1 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.

To disable, enable, or change the evaluation edition or unusable editions, use the `ALTER MATERIALIZED VIEW` statement.

To display the evaluation editions and unusable editions of materialized views, use the static data dictionary views `*_MVIEWS` .

Dropping the evaluation edition invalidates the materialized view. Dropping an edition where the materialized view is usable does not invalidate the materialized view.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database SQL Language Reference* for more information about `CREATE MATERIALIZED VIEW` statement
>
> *Oracle Database SQL Language Reference*
>
> *Oracle Database Reference*

33.2.1.2.2 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 Materialized Views.

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

To display the evaluation editions of virtual columns, use the static data dictionary views `*_TAB_COLS` .

> **See Also**
>
> - *Oracle Database SQL Language Reference* for the complete syntax and semantics of the virtual column definition
> - *Oracle Database Reference* for more information about `ALL_TAB_COLS`

33.2.1.3 Editionable 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 and are shown by the dynamic performance view `V$EDITIONABLE_TYPES` .

If the value of `COMPATIBLE` is 12 or greater, then these schema object types are editionable in the database:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

See Also

- *Oracle Database Administrator's Guide* for more information about COMPATIBLE initialization parameter
- *Oracle Database Reference* for more information about V$EDITIONABLE_TYPES
- Enabling Editions for a User

33.2.1.4 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 . 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 or USER_USERS .

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 ]... ]
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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 VIEW, SYNONYM, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, TYPE, TYPE BODY, and LIBRARY.

To enable edition for other object types that are not enabled by default, you must explicitly specify the object type in the FOR clause. For example, to enable editions for SQL TRANSLATION PROFILE, run the following command:

```
ALTER USER user ENABLE EDITIONS [ FOR SQL TRANSLATION PROFILE];
```

The static data dictionary view DBA_EDITIONED_TYPES lists all the object types that can appear in the FOR type clause. Refer to this view if you need to enable editions for specific object types.

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 EDITIONABLE and NONEDITIONABLE Properties.

Topics:

- Potentially Editioned Objects with Noneditioned Dependents
- Users Who Cannot Have Editions Enabled

See Also

Oracle Database SQL Language Reference for the complete syntax and semantics of the CREATE USERand ALTER USER statements

Oracle Database Reference for more information about DBA_EDITIONED_TYPES

Oracle Database Reference for more information about DBA_USERS

Oracle Database Reference for more information about USER_USERS

33.2.1.4.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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
   ```

`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.

> **See Also**
>
> - *Oracle Database PL/SQL Language Reference* for information about the `ALTER` statements for PL/SQL objects
> - *Oracle Database SQL Language Reference* for information about the `ALTER` statements for SQL objects
> - Invalidation of Dependent Objects for information about invalidation of dependent objects

33.2.1.4.2 Users Who Cannot Have Editions Enabled

You cannot enable editions for these users:

- Oracle-maintained users

  For an Oracle-maintained user, the value of the column ORACLE_MAINTAINED is Y in the *_USERS views.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.)

See Also

- *Oracle Database Administrator's Guide* for information about common users in a CDB
- *Oracle Database PL/SQL Language Reference* for the syntax of the `ALTER TYPE RESET` statement

33.2.1.5 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` or `*_OBJECTS_AE` .

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 AI 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
- *Oracle Database Reference* for more information about `*_OBJECTS`
- *Oracle Database Reference* for more information about `*_OBJECTS_AE`

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.2.1.5.1 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 `NONEDITIONABLE` 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. 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 `NONEDITIONABLE` 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 Also

- Current Edition and Session Edition for information about the current edition
- Example: Dropping an Editioned Object
- Example: Creating an Object with the Name of a Dropped Inherited Object

33.2.1.5.2 Replacing or Altering EDITIONABLE and NONEDITIONABLE 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 `NONEDITIONABLE`, 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 `NONEDITIONABLE`, or the reverse.

Altering an editioned object is a live operation with respect to the editions in which the altered object is invisible.

33.2.1.6 Rules for Editioned Objects

- A noneditioned object usually cannot depend on an editioned object .
- An Abstract Data Type (ADT) cannot be both editioned and evolved.

Copyright © 1999, 2025, Oracle and/or its affiliates.

You cannot evolve an editioned ADT. The `ALTER TYPE` statement fails with ORA-22348 if either of the following is true:

- The type is an editioned ADT and the `ALTER TYPE` statement has no *`type_compile_clause`*. You can use the `ALTER TYPE` statement to recompile an editioned object type, but not for any other purpose.
- The type has a dependent that is an editioned ADT and the `ALTER TYPE` statement has a `CASCADE` clause.

Refer to *Oracle Database PL/SQL Language Reference* for more information on the *`type_compile_clause`* and the `CASCADE` clause.

- 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.

See Also

- Name Resolution for Editioned and Noneditioned Objects
- *Oracle Database Object-Relational Developer's Guide* for information about type evolution

33.2.2 Creating an Edition

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: Editioned Objects and Copy-on-Change 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
- *Oracle Database Administrator's Guide* for information about CDBs

33.2.3 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

33.2.3.1 Example: Editioned Objects and Copy-on-Change

Example 33-1 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 33-1 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;
```

5. Invoke the procedure:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.
```

> **See Also**
>
> Changing Your Session Edition for information about `ALTER SESSION SET EDITION`

33.2.3.2 Example: Dropping an Editioned Object

Example 33-2 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.

Because `e2` dropped the procedure `goodbye`:

- Its descendents do not inherit the procedure `goodbye`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- No object named goodbye is visible in e2, so e2 can create an object named goodbye, but it must be an editioned object. If e2 creates a new editioned object named goodbye, then the descendents of e2 inherit that object.

Example 33-2 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!');
  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.

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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.
```

See Also

- *Oracle Database PL/SQL Language Reference*, for more information about the DROP PROCEDURE statement, including the privileges required to use it
- Creating New EDITIONABLE and NONEDITIONABLE Objects
- Changing Your Session Edition for more information about ALTER SESSION SET EDITION

33.2.3.3 Example: Creating an Object with the Name of a Dropped Inherited Object

In Example 33-3, 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 33-3 Creating an Object with the Name of a Dropped Inherited Object

1. Return to e2:

```
ALTER SESSION SET EDITION = e2;
```

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 editioned.

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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.
```

See Also

- Changing Your Session Edition for information about `ALTER SESSION SET EDITION`
- Enabling Editions for a User for instructions to enable editions for a user

33.2.4 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`.

See Also

Oracle Database SQL Language Reference for information about the `GRANT` statement

33.2.5 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
  ```

- Make the edition the database default edition:

  ```
  ALTER DATABASE DEFAULT EDITION = edition_name
  ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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`.

See Also

- *Oracle Database SQL Language Reference* for information about the `ALTER DATABASE` statement
- *Oracle Database SQL Language Reference* for information about the `GRANT` statement

33.2.6 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

33.2.6.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

As of Oracle Database 11*g* 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 11*g* 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

33.2.6.2 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: Editioned Objects and Copy-on-Change and Example: Dropping an Editioned Object 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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

33.2.6.3 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

33.2.6.4 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.
- You run a statement by calling the `DBMS_SQL.PARSE` procedure, specifying the edition in which the statement is to run, as in Example 33-4.

  While the statement is running, the current edition is the specified edition, but the session edition does not change.

Example 33-4 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_SQL.PARSE` procedure.

Example 33-4 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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_SQL.PARSE (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.
```

See Also

- Crossedition Trigger Interaction with Editions
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SQL.PARSE` procedure

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.2.6.5 Disassociating a Session from an Edition

To disassociate a session from an edition at the end of the current client call, use the `DBMS_SESSION` package's `USE_DEFAULT_EDITION_DEFERRED` procedure.

To disassociate a session from an edition at the end of each and every client call, use the `DBMS_SESSION` package's `USE_DEFAULT_EDITION_ALWAYS` procedure.

When the `USE_DEFAULT_EDITION_DEFERRED` or `USE_DEFAULT_EDITION_ALWAYS` procedure is used, no edition is used by the session after the current call and before the next top-level call. The session uses the current database's default edition on its next operation.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about the USE_DEFAULT_EDITION_DEFERRED procedure and the USE_DEFAULT_EDITION_ALWAYS procedure

33.2.7 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
> ```

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 AND TYPE = 'EDITION'
/
```

When you retire an edition, update the evaluation editions and unusable editions of noneditioned objects accordingly.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - Noneditioned Objects That Can Depend on Editioned Objects for information about changing evaluation editions and unused editions
> - *Oracle Database SQL Language Reference* for information about the `REVOKE` statement
> - *Oracle Database SQL Language Reference* for information about the `ALTER DATABASE` statement

33.2.8 Dropping an Edition

> **Note**
>
> If the edition includes crossedition triggers, see Dropping the Crossedition Triggers, before you drop the edition.

To drop an edition, use the `DROP EDITION` statement. 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.
- 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.

If the edition is the root, and the `COMPATIBLE` parameter is set to 12.2.0 or higher, the edition is marked as unusable. A covered object is an editioned object that is no longer inherited in any usable descendent edition. Each covered object in any unusable edition is dropped by an automated scheduled maintenance process. After there is no object left in the root unusable edition, the edition itself is dropped automatically. This cleanup process is repeated for each unusable root edition found. A user can run this process on demand by manually executing the `DBMS_EDITIONS_UTILITIES.CLEAN_UNUSABLE_EDITIONS` procedure (see *Oracle Database PL/SQL Packages and Types Reference*).

If the `COMPATIBLE` is set to 12.1.0 or lower, the root edition must have no objects that its descendents inherit. Each object inherited from the root edition must either be actualized or dropped explicitly before the edition can be dropped.

If the edition is the leaf, every editioned object in the leaf is dropped, followed by the edition itself before the statement finishes execution.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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 Changing Your Session Edition.

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
   ```

When you drop an edition, update the evaluation editions and unusable editions of noneditioned objects accordingly.

> **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
> - *Oracle Database SQL Language Reference* for more information about the `DROP EDITION` statement
> - *Oracle Database PL/SQL Language Reference* for information about the `ALTER` statements for PL/SQL objects
> - *Oracle Database SQL Language Reference* for information about the `ALTER` statements for SQL objects.
> - Noneditioned Objects That Can Depend on Editioned Objects for information about changing evaluation editions and unused editions

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.3 Editions and Audit Policies

Audit policies configured on an object in an edition is applied across all the editions of the object. Existing audit policies are extended to the objects in newly created editions.

Auditing is the monitoring and recording of configured database actions from both database users and application users, who are recognized in the database using the `CLIENT_IDENTIFIER` attribute. You can audit SQL statements, security-relevant activities and also fine-grained activities.

> **See Also**
>
> *Oracle Database Security Guide* for information on audit policies.

33.4 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."

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Replacing an Editioning View
- Dropped or Renamed Base Tables
- 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`.

33.4.1 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.

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 `NONEDITIONABLE`, 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
> - Enabling Editions for a User

33.4.2 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

- INSERT
- SELECT
- UPDATE

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about referring to partitioned tables

33.4.3 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

33.4.4 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.

33.4.5 Dropped or Renamed Base Tables

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.

33.4.6 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- Guidelines for Managing Indexes
- *Oracle Database SQL Language Reference* for information about `INDEX` hints
- SQL Optimizer Index Hints

33.4.7 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

33.5 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.5.1 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.

33.5.2 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.

33.5.3 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.

Topics:

- Which Triggers Are Visible
- What Kind of Triggers Can Fire
- Firing Order
- Crossedition Trigger Execution

> **See Also**
>
> When the Current Edition Might Differ from the Session Edition

33.5.3.1 Which Triggers Are Visible

Editions inherit noncrossedition triggers in the same way that they inherit other editioned objects (see 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.5.3.2 What Kind of Triggers Can Fire

What kind of triggers can fire depends on the category of the triggering DML statement.

Categories:

- Forward Crossedition Trigger SQL
- Reverse Crossedition Trigger SQL
- Application SQL

> **Note**
>
> The `APPEND` hint on a SQL `INSERT` statement does not prevent crossedition triggers from firing.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about the `APPEND` hint

33.5.3.2.1 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_SQL.PARSE` 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.

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_SQL.PARSE` 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.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SQL.PARSE` procedure

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.5.3.2.2 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.

33.5.3.2.3 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.
- 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.

> **See Also**
>
> *Oracle Database PL/SQL Language Reference* for information about `DBMS_SQL` package

33.5.3.3 Firing Order

For a trigger to fire in response to a specific DML statement, the trigger must:

- Be the right kind
- 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Database PL/SQL Language Reference* for general information about trigger firing order
> - What Kind of Triggers Can Fire

33.5.3.3.1 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.

33.5.3.3.2 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 descendents of the current edition, in the order that the descendents 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)

33.5.3.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.5.4 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 Enabling Editions for a User.)

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 or a native dynamic SQL statement .
- 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
- *Oracle Database PL/SQL Language Reference* for more information about Static SQL
- *Oracle Database PL/SQL Language Reference* for more information about native dynamic SQL

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.5.4.1 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 (do a bulk upgrade of rows from old format to new format)
- How many times the body will run for each old row.

Topics:

- Handling Data Transformation Collisions
- Handling Changes to Other Tables

> **See Also**
>
> Transforming Data from Pre- to Post-Upgrade Representation for information about applying transforms

33.5.4.1.1 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 you are applying the transform in bulk, and `FALSE`, if the trigger is running because of a serendipitous change. (`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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 33-5 creates a crossedition trigger that uses the `APPLYING_CROSSEDITION_TRIGGER` function and the `IGNORE_ROW_ON_DUPKEY_INDEX` and `CHANGE_DUPKEY_ERROR_INDEX` hints 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);
```

> **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

Example 33-5 Crossedition Trigger that Handles Data Transformation Collisions

```
CREATE OR REPLACE TRIGGER trigger1
  BEFORE INSERT OR UPDATE ON table1
  FOR EACH ROW
  CROSSEDITION
DECLARE
  row_already_present  EXCEPTION;
  PRAGMA EXCEPTION_INIT(row_already_present, -38911);
BEGIN
  IF APPLYING_CROSSEDITION_TRIGGER THEN
    /* The trigger is running because of applying the transform.
       If the old edition of the app has already caused this trigger
          to insert a row, we do not modify the row as part of applying
          the transform. Therefore, insert the new row into table2 only if
          it is not 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
     /* The trigger is running because of a serendipitous change.
          If no previous run of the trigger has already inserted
          the corresponding row into table2, insert the new row;
          otherwise, update the previously inserted 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    END;
  END IF;
END;
/
```

33.5.4.1.2 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.

33.5.5 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.

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 first two ways of applying the transform, invoke either the `DBMS_SQL.PARSE` 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

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_SQL.PARSE` 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_SQL.PARSE` procedure
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_PARALLEL_EXECUTE` package
- Forward Crossedition Triggers for information about forward crossedition triggers
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_EDITIONS_UTILITIES.SET_NULL_COLUMN_VALUES_TO_EXPR` procedure

33.5.5.1 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 `DBMS_UTILITY.WAIT_ON_PENDING_DML` procedure.
3. Apply the transform.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **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.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_UTILITY.WAIT_ON_PENDING_DML` procedure

33.5.6 Dropping the Crossedition Triggers

To drop a crossedition trigger, use the `DROP TRIGGER` statement. 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.

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. .

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle Database PL/SQL Language Reference* for more information about `DROP TRIGGER` statement
- Dropping an Edition for more information about dropping editions
- *Oracle Database SQL Language Reference* for more information about `ALTER TABLE` statement

33.6 Displaying Information About EBR Features

Topics:

- Displaying Information About Editions
- Displaying Information About Editioning Views
- Displaying Information About Crossedition Triggers

33.6.1 Displaying Information About Editions

Table 33-1 briefly describes the static data dictionary views that display information about editions.

Table 33-1 *_ 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> `*_OBJECTS` and `*_OBJECTS_AE` include dependent objects that are invalidated by operations in Table 32-2 only after one of the following:
>
> - A reference to the object (either during compilation or execution)
> - An invocation of `DBMS_UTILITY.COMPILE_SCHEMA`
> - An invocation of any `UTL_RECOMP` subprogram

> **See Also**
>
> - *Oracle Database Reference* for more information about a specific view
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_UTILITY.COMPILE_SCHEMA` procedure
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about `UTL_RECOMP` subprogram

33.6.2 Displaying Information About Editioning Views

Table 33-2 briefly describes the static data dictionary views that display information about editioning views.

Table 33-2 *_ 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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

> **See Also**
>
> *Oracle Database Reference* for more information about a specific view

33.6.3 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.

> **See Also**
>
> *Oracle Database Reference* for information about V$SQL_SHARED_CURSOR

33.7 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 Enabling Editions for a User.

2. Prepare your application to use editioning views.

   For instructions, see Preparing Your Application to Use Editioning Views.

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 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 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 Procedure for EBR Using Crossedition Triggers.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

33.7.1 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Any triggers that depend on renamed columns are now invalid. For details, see the entry for `ALTER TABLE table RENAME column` in Table 32-2.

3. Create the editioning view, giving it the original name of the table.

   For instructions, see Creating an Editioning View.

   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 Enabling Editions for a User).

9. Notify the owners of private synonyms that refer to the table that they must re-create those synonyms.

33.7.2 Procedure for EBR Using Only Editions

Use this procedure only if every object that you will redefine is editioned (as defined in Editioned and Noneditioned Objects). Tables are never editioned objects.

1. Create a new edition.

   For instructions, see Creating an Edition.

2. Make the new edition your session edition.

   For instructions, see Changing Your Session Edition.

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 step3) or roll back the application upgrade (for instructions, see Rolling Back the Application Upgrade).

6. Make the new edition (the upgraded application) available to all users.

   For instructions, see Making an Edition Available to All Users.

Copyright © 1999, 2025, Oracle and/or its affiliates.

7. Retire the old edition (the original application), so that all users except SYS use only the upgraded application.

   For instructions, see Retiring an Edition.

   Example 33-6 shows how to use the preceding procedure to change a very simple PL/SQL procedure.

Example 33-6 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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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;
```

33.7.3 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 Creating an Edition.

2. Make the new edition your session edition.

   For instructions, see Changing Your Session Edition.

3. In the new edition, if the editioning views are read-only, make them read-write.

   For instructions, see Changing the Writability of an Editioning View.

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 Rolling Back the Application Upgrade).

8. Make the upgraded application available to all users.

   For instructions, see Making an Edition Available to All Users.

9. Retire the old edition (the original application), so that all users except `SYS` use only the upgraded application.

   For instructions, see Retiring an Edition.

Copyright © 1999, 2025, Oracle and/or its affiliates.

33.7.4 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 Creating an Edition.

2. Make the new edition your session edition.

   For instructions, see Changing Your Session Edition.

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 32-2.

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 Creating a Crossedition Trigger.

   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 Transforming Data from Pre- to Post-Upgrade Representation.

   > **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 Rolling Back the Application Upgrade).

10. (Optional) Grant the `USE` privilege on your session edition to the early users of the upgraded application.

    For instructions, see Making an Edition Available to Some Users.

Copyright © 1999, 2025, Oracle and/or its affiliates.

11. Make the upgraded application available to all users.

    For instructions, see Making an Edition Available to All Users.

12. Disable or drop the constraints and then drop the crossedition triggers.

    For instructions, see Dropping the Crossedition Triggers.

13. Retire the old edition (the original application), so that all users except SYS use only the upgraded application.

    For instructions, see Retiring an Edition.

33.7.5 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 Changing Your Session Edition.

2. Drop the new edition that you created for the upgrade.

   For instructions, see Dropping an Edition.

3. If you created new table columns during the upgrade, reclaim the space that they occupy (for instructions, see Reclaiming Space Occupied by Unused Table Columns).

33.7.6 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*).

33.7.7 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> Before you can use EBR to upgrade an application, you must enable editions for every schema that the application uses. For instructions, see Enabling Editions for a User.

33.7.7.1 Existing Application

The existing application—the application to be upgraded—consists of a single table on which a trigger is defined.

The existing application has a trigger, which you can check. The following examples show the existing application:

- Example: Creating the Existing Application
- Example: Viewing Data in Existing Table

33.7.7.1.1 Example: How the Existing Application Was Created

The application was created as in Example 33-7.

Example 33-7 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;
/
```

33.7.7.1.2 Example: Viewing Data in the Existing Table

Example 33-8 shows how the table `Contacts` looks after being populated with data.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 33-8 Viewing Data in the 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.

33.7.7.2 Preparing the Application to Use Editioning Views

Example 33-9 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 33-9 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;
   ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

3. Create editioning view:

```
CREATE OR REPLACE EDITIONING VIEW Contacts AS
  SELECT
    ID                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;
/
```

33.7.7.3 Using EBR to Upgrade the Example Application

You can use triggers to upgrade an existing application.

The following examples show how to use triggers to updated the example application:

- Example: Creating Edition in Which to Upgrade the Example Application
- Example: Changing the Table and Replacing the Editioning View
- Example: Creating and Enabling the Crossedition Triggers
- Example: Applying the Transforms
- Example: Viewing Data in the Changed Table

33.7.7.3.1 Example: Creating an Edition in Which to Upgrade the Example Application

Example 33-10 shows how to create an edition in which to upgrade the existing application (in Existing Application), make the new edition the session edition, and check that the new edition really is the session edition.

Example 33-10 Creating an Edition in Which to Upgrade the Example Application

1. Create the 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.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

In the `Post_Upgrade` edition, Example: Creating an Edition in Which to Upgrade the Example Application 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."

33.7.7.3.2 Example: Changing the Table and Replacing the Editioning View

In the `Post_Upgrade` edition, Example 33-11 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 33-11 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;
```

33.7.7.3.3 Example: Creating and Enabling the Crossedition Triggers

In the `Post_Upgrade` edition, Example 33-12 shows how to apply the transforms.

Example 33-12 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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;

    RETURN TRUE;
  END Is_US_Number;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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, '-', '.')
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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;
/
```

See Also

Oracle Database PL/SQL Packages and Types Reference for information about the `DBMS_UTILITY.WAIT_ON_PENDING_DML` procedure

33.7.7.3.4 Example: Applying the Transforms

In the Post_Upgrade edition, Example 33-13 example shows how to apply the transforms.

Example 33-13 Applying the Transforms

```
DECLARE
  c NUMBER := DBMS_SQL.OPEN_CURSOR();
  x NUMBER;
BEGIN
  DBMS_SQL.PARSE(
    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;
/
```

33.7.7.3.5 Example: Viewing Data in the Changed Table

In the `Post_Upgrade` edition, Example 33-14 shows how to check that the change worked as intended. Compare Example: Viewing Data in the Changed Tableto Example: Viewing Data in the Existing Table.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 33-14 Viewing Data in the 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 Procedure for EBR Using Crossedition Triggers.

Copyright © 1999, 2025, Oracle and/or its affiliates.

34
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 commit 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 is used by Application Continuity for automatic and transparent transaction replay.

Transaction Guard provides these benefits:

- Preserves the returned outcome - committed or uncommitted so that it can be relied on
- Ensures a known commit outcome for every transaction
- Can be used to provide at-most-once transaction execution for applications that wish to resubmit themselves
- Is used by Application Continuity for automatic and transparent transaction replay

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
- Transaction Guard Support during DBMS_ROLLING Operations

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

34.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 canceled 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.

34.2 Solution That Transaction Guard Provides

Effective with Oracle Database 12*c* 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 12*c* Release 1 (12.1.0.1) or later). This includes idempotence for transactions executed using autocommit, from inside PL/SQL, from remote transactions, One-Phase XA 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 infrastructure. This includes Oracle Real Application Clusters (Oracle RAC) and RAC One, Data Guard.
- Service name uniqueness across global databases and across databases that are consolidated into a Multitenant infrastructure. This ensures that connections are properly directed to the transaction information.

34.3 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)

34.3.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

34.3.2 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 terminated session. This is needed to determine the status of this last transaction.

The `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL subprogram 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.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL subprogram

34.3.3 Transaction Guard Coverage

You may use Transaction Guard on each database in your system, including restarting on and failing over between single instance database, Real Application Clusters, Data Guard and Active Data Guard.

Transaction Guard is supported with the following Oracle Database configurations:

- Single Instance Oracle RDBMS
- Real Application Clusters
- Data Guard
- Active Data Guard

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Multitenant including unplug/plug and relocates across the PDB/CDB, but excludes "with clone" option
- Global Data Services for the above database configurations

Transaction Guard supports the following transaction types against Oracle Database:

- 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
- XA transactions using One Phase Optimizations including XA commit flag TMONEPHASE and read optimizations
- `ALTER SESSION SET` Container with Service clause, where the service uses Transaction Guard

Transaction Guard supports the following client drivers :

- 12*c* JDBC type 4 driver
- 12*c* OCI and OCCI client drivers
- 12*c* Oracle Data Provider for .NET (ODP.NET), Unmanaged Driver
- 12*c* ODP.NET, Managed Driver in ODAC 12*c* Release 4 or higher

34.3.4 Transaction Guard with XA Transactions

Starting with Oracle Database 12.2 Release, Transaction Guard supports XA transactions to determine the outcome of one phase transactions. Transaction Guard supports local transactions and XA transactions that use TMONEPHASE during the `commit` operation. When the application issues an XA transaction that uses `TMTWOPHASE`, the Transaction Guard disables itself for that transaction and automatically re-enables to prepare itself for the next transaction. This allows Transaction Guard to support the following XA transactions:

1. Local transactions that use `autocommit`
2. Local transactions that use an explicit `commit`
3. XA transactions that commit with `TMONEPHASE` flag

TP Monitors and Applications can use Transaction Guard to obtain the outcome of `commit` operation for these transaction types. Transaction Guard disables itself for externally-managed `TMTWOPHASE` commit operations and automatically re-enables for the next transaction. If the Transaction Guard APIs are used with a `TMTWOPHASE` transaction, a warning message is returned as Transaction Guard is disabled. The TP monitors own the commit outcome for `TMTWOPHASE` transactions. This functionality allows TP monitors to return an unambiguous outcome for `TMONEPHASE` operations.

Copyright © 1999, 2025, Oracle and/or its affiliates.

34.3.5 Transaction Guard Exclusions

Transaction Guard intentionally excludes recursive transactions and autonomous transactions so that they can be re-executed.

As of Oracle Database 12*c* Release 2, Transaction Guard also excludes:

- Two Phase XA transactions are managed externally. When using XA transactions, Transaction Guard maintains the commit outcome for one-phase XA transactions, and silently disables itself for externally-managed two-phase transactions because this outcome is owned by the TP monitor.
- Active Data Guard with read/write database links for forwarding transactions
- Golden Gate and Logical Standby for determining the outcome when failing across logical databases. Golden Gate and Logical Standby endpoints can use Transaction Guard
- Full database import cannot be executed with Transaction Guard enabled. Use an admin service without Transaction Guard for full database imports. User and object imports are not excluded.
- TAF and Application Continuity handle Transaction Guard internally. Do not code Transaction Guard in your application in the following places:
  - A failed return from TAF
  - TAF Callback for TAF or for Application Continuity for OCI and ODP.NET
  - JDBC initialization callback for Application Continuity for Java

Transaction Guard excludes failover across databases maintained by replication technology:

- Replication to Golden Gate
- Replication to Logical Standby
- PDB clones clause (excluding PDB online relocation 12c Release 2 and later)
- All third party replication solutions

If you are using a database replica using any replication technology such as Golden Gate, or Logical Standby, or 3rd party replication, you may not use Transaction Guard between the primary and the secondary databases in this configuration.

You may use Transaction Guard on each database that participates in the replication. In this case, each database must use a different database unique identifier. Use V$DATABASE to obtain the DBID for each database.

34.4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

34.4.1 Configuration Checklist

To use Transaction Guard with an application, you must do the following:

- Use Oracle Database 12*c* 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>;
  ```
- Increase `DDL_LOCK_TIMEOUT` if using Transaction Guard with DDL statements..

To use Transaction Guard with an application, Oracle recommends that you do the following:

- Locate and define the transaction history table for optimal performance.
- 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).

> **See Also**
>
> - *Oracle Database Reference* for more information about `DDL_LOCK_TIMEOUT`
> - Transaction History Table

34.4.2 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 );
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for information about the `ALTER TABLE` statement

34.4.3 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` 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 database links that forward DMLs 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.

> **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 PL/SQL Packages and Types Reference* for more information about `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure

34.4.3.1 Example: Adding and Modifying a Service for a Server Pool

If you are using Oracle RAC or Oracle RAC One, then use the `srvctl` command to create and modify services.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 34-1 shows the use of `srvctl`. You can also use Global Data Services (GDSCTL).

Example 34-1 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
```

34.4.3.2 Example: Adding an Administrator-Managed Service

If you are using Oracle RAC or Oracle RAC One, then use the `srvctl` command to create and modify services.

Example 34-2 shows the use of `srvctl`. You can also use Global Data Services (GDSCTL)

Example 34-2 Adding an Administrator-Managed Service

```
srvctl add service -database codedb -service GOLD -preferred serv1 -available serv2 -
commit_outcome TRUE  -retention 604800
```

34.4.3.3 Example: Modifying a Service (PL/SQL)

If you are using a single-instance database, use the `DBMS_SERVICE.MODIFY_SERVICE` PL/SQL procedure to modify services and use FAN.

Example 34-3 modifies a service (but substitute the actual service name for `<service-name>`).

Example 34-3 Modifying a Service (PL/SQL)

```
DECLARE
  params dbms_service.svc_parameter_array;
BEGIN
  params('COMMIT_OUTCOME'):='true';
  params('RETENTION_TIMEOUT'):=604800;
  params('aq_ha_notifications'):='true';
  dbms_service.modify_service('<service-name>',params);
END;
/
```

34.5 Developing Applications That Use Transaction Guard

To use Transaction Guard, review the requirements and recommendations in Configuration Checklist, and follow these steps in the error handling when a recoverable error occurs:

> **Note**
>
> If you are using TAF, skip to Transaction Guard and Transparent Application Failover.

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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

34.5.1 Typical Transaction Guard Usage

The following pseudocode shows a typical usage of Transaction Guard:

```
Receive a FAN down event (or recoverable error)

FAN cancels the terminated session

If recoverable error  (new OCI_ATTRIBUTE for OCI, isRecoverable for JDBC)
     Get last LTXID from terminated 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
```

34.5.2 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 34-1 shows several conditions or situations that require some LTXID-related action, and for each the application action and next LTXID to use.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 34-1 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 `GET_LTXID_OUTCOME` to determine the transaction status.	Application takes a new connection (with its own LTXID-B 0) and calls `GET_LTXID_OUTCOME` with the LTXID of the last failed session (LTXID-A).	New LTXID-B 0 Also set using the JDBC callback when registered
Application finds that the last session transaction status is `COMMITTED` and `USER_CALL_COMPLETED`.	Returns committed status to client; the application may be able to continue.	(Not applicable)
Application finds that the last session transaction status is `COMMITTED` and NOT `USER_CALL_COMPLETED`.	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 `NOT COMMITTED`.	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. If the new request executes any commits, server returns commit messages with LTXID-B 2 and increasing.	New LTXID-B 2 .. N Also set using the JDBC callback when registered
Application receives a recoverable error if it has decided to replay.	Application takes a new connection (with LTXID-C 0) and calls `GET_LTXID_OUTCOME` with the LTXID of LAST session (LTXID-B N).	LTXID-C 0 on the new session. 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 `GET_LTXID_OUTCOME` again with the LTXID of LAST session (LTXID-C N).	LTXID-D 0 on the new session. Also set using the JDBC callback when registered

34.5.3 Transaction Guard and Transparent Application Failover

When Transparent Application Failover (TAF) is enabled with Transaction Guard, TAF handles the errors for developers. Do not code Transaction Guard when you are using TAF because it has embedded the Transaction Guard code starting with Oracle Database 12*c* Release 1 (12.1.0.1). When both TAF and Transaction Guard are used, developers can use the following TAF errors to rollback and safely resubmit, or return uncommitted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- ORA-25402
- ORA-25408
- ORA-25405

Developers must not use GET_LTXID_OUTCOME procedure directly when TAF is enabled because TAF is already processing Transaction Guard.

> **Note**
>
> TAF is not invoked on session failure (this includes “kill -9” at operating system level, or ALTER SYSTEM KILL session). TAF is invoked on the following conditions:
>
> - INSTANCE failure
> - FAN NODE DOWN event
> - SHUTDOWN transactional
> - Disconnect POST_TRANSACTION

34.5.4 Using Transaction Guard with ODP.NET

The following rules apply to using Transaction Guard with ODP.NET:

- The LTXID is not available after promoting to XA in both the ODP.NET providers.
- Starting with Oracle Database 12c Release 2 (12.2.0.1), ODP.NET handles Transaction Guard for application based on its availability and handling abilities. When using ODP.NET, the LTXID is exposed to the application only when ODP.NET is unable to obtain the commit outcome on behalf of the application. For example, during an extended failover to Data Guard.
- Developers must not code Transaction Guard in the TAF callback or JDBC initialization callback. Transaction Guard is handled for you.

34.5.5 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 (client-side session). 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 request
- Allows to cancel (real Cancel operation and not **Ctrl-C**) timed out requests, and optional re-submission by the application

34.5.6 Improved Commit Outcome for XA One Phase Optimizations

Copyright © 1999, 2025, Oracle and/or its affiliates.

Starting with Oracle Database 12*c* Release 2 (12.2.0.1), Transaction Guard is used with Transaction Processing Monitors (TPM) to determine the outcome of a commit operation when using one-phase optimizations (TMONEPLHASE flag). The Transaction Guard uses the `GET_LTXID_OUTCOME` package to help the TPM to determine if the connection to the resource manager is lost or if an ambiguous error is returned.

Table 34-2 Transaction Manager Conditions/ Situations and Actions

Condition or Situation	Transaction Manager Action
`Commit` has not been issued, and if the transaction has rolled back.	Transaction Manager returns a `rollback`.
`Commit` has been issued and if an ambiguous result is returned.	Transaction Manager can use Transaction Guard (XA) to determine the outcome when the error is recoverable.
If the transaction is `COMMITTED`.	Transaction Manager returns `COMMITTED`.
If the transaction is `UNCOMMITTED`.	The Transaction Manager borrows a new connection and reissues the `COMMIT`. The original `LTXID` is blocked by calling `GET_LTXID_OUTCOME`.

34.5.7 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 terminated.

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.
- Do not code Transaction Guard if the application is using TAF. Use the new TAF error codes to return the results instead.

> **Note**
>
> This rule does not apply to Application Continuity.

> **See Also**
>
> Transaction Guard and Transparent Application Failover for more information about TAF

Copyright © 1999, 2025, Oracle and/or its affiliates.

34.6 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 a solution that does not require coding, configure your application to use Application Continuity. For developing your own replay, the application developer codes using Transaction Guard. You can have an application coded for both Transaction Guard and Application Continuity. The Application Continuity takes effect first and the custom Transaction Guard code takes effect only when the Application Continuity is unable to replay. It is not required to use both, but, they are compatible if an application uses both Transaction Guard and Application Continuity. If an application wishes to add Transaction Guard API's in addition to Application Continuity, Transaction Guard can return the commit outcome when replay is disabled or unsuccessful.

See Also

- *Oracle Real Application Clusters Administration and Deployment Guide* for information about Transaction Guard and Application Continuity with Oracle RAC
- *Oracle Database JDBC Developer's Guide* for information about connecting to the database with JDBC
- *Oracle Call Interface Programmer's Guide* for information about connecting to the database with Oracle Call Interface (OCI)
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about ODP.Net Driver

34.7 Transaction Guard Support during DBMS_ROLLING Operations

Transaction Guard ensures continuous application operation during the rolling upgrade operations performed on Oracle Database.

The rolling upgrade (`DBMS_ROLLING`) procedure is used when Oracle Database requires a major release upgrade. During the `DBMS_ROLLING` procedure, the upgrades are done on a logical standby while the primary database remains open for production.

34.7.1 Rolling Upgrade Using Transient Logical Standby

The transient logical process used in the rolling upgrade begins and ends with a physical standby database while temporarily being converted to a logical standby database for the upgrade. A Transient Logical Standby is used for major upgrades to ensure that user applications do not fail during the major upgrades.

Copyright © 1999, 2025, Oracle and/or its affiliates.

A rolling upgrade using Transient Logical Standby:

- Temporarily converts an existing physical standby to a logical standby database for the duration of the upgrade
- Executes a rolling upgrade on the logical standby database to release 'n+1' while the production runs on the primary database at release 'n'
- Returns the logical standby back to its original status as a physical standby database after the upgrade is successful
- Resynchronizes the physical standby with the primary database using SQL apply
- Performs a switchover to transition the physical standby to the production role running on the new release. The physical standby becomes the new primary database
- Converts original primary database back to physical standby database and resynchronizes with the new primary, automatically completing the upgrade process
- Upgrades the physical standby using the redo stream

34.7.2 Transaction Guard Support During Major Database Version Upgrades

Starting with Oracle AI Database 26ai, Transaction Guard works during `DBMS_ROLLING` operations to ensure continuous application functions during switchover, issued by `DBMS_ROLLING` to Transient Logical Standby.

Transaction Guard returns the commit outcome of the current in-flight transaction when an error or outage occurs. Applications embed the Transaction Guard APIs in their error handling procedures to ensure that work continues without any in-flight work lost or duplicate submissions after an outage. Transaction Guard provides idempotence support to ensure that a commit occurs not more than once when a transaction is re-processed (replay) after an outage.

Transaction Guard ensures continuous application operation during the `DBMS_ROLLING` switchover operation to Transient Logical Standby. Transaction Guard ensures that the last commit outcome of transactions in the in-flight sessions during a switchover outage is used to protect the applications from duplicate submissions of the transactions on replay.

Transaction Guard maintains a transaction history table called `LTXID_TRANS` that has the mapping of logical transaction identifiers (`LTXID`s) to database transactions. For a failover to succeed after an outage, the changes to `LTXID_TRANS` from the primary database must first replicate and apply to Transient Logical Standby. With supplemental logging enabled for the `DBMS_ROLLING` procedure, Transaction Guard uses SQL to allow supplemental capture of `LTXID_TRANS` at CDB and PDB levels. The capture process replicates the `LTXID_TRANS` table and the apply process reads and recreates the `LTXID_TRANS` tables for the logical standby, along with the committed user transactions.

As a part of its support for the `DBMS_ROLLING` procedure, Transaction Guard performs the following functions:

- Tracks when the primary database is in `DBMS_ROLLING` mode (when the database upgrade is initiated)
- Checks that supplemental logging is in use
- Records the redo vector for the primary key (PK) at runtime while in supplemental logging mode

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Waits for all current updates to finish and replicate to the logical standby before performing the LTXID replication
- Replicates the `LTXID_TRANS` tables and applies the redo to Transient Logical Standby for each PDB
- Provides a mechanism for failover to know about successful LTXID replication
- Enforces last commit outcome for inflight sessions on replay after an outage
- Handles new users during supplemental capture and apply process to ensure that any apply does not create mismatched logged-in UIDs (user IDs) at the target database

Copyright © 1999, 2025, Oracle and/or its affiliates.

35
Table DDL Change Notification

This chapter explains how to use Table DDL Change Notification for receiving notifications about DDL changes in database tables.

Topics:

- Overview of Table DDL Change Notification
- Table DDL Change Notification Terminology
- Benefits of Table DDL Change Notification
- Features of Table DDL Change Notification
- Using Table DDL Change Notification
- Registering for Table DDL Change Notification
- Unregistering for Table DDL Change Notifications
- Supported DDL Events and Commands
- Monitoring Table DDL Change Notification

35.1 Overview of Table DDL Change Notification

Table DDL Change Notification provides Oracle Call Interface (OCI) clients an efficient mechanism to subscribe for DDL notifications on tables of interest. Table DDL Change Notification ensures that applications are notified when DDL statements make changes to a table. Changes are captured as DDL events and these events are processed asynchronously without blocking the user DML activity.

Applications seeking table metadata can use Table DDL Change Notification instead of continuously polling for DDL changes, and benefit from reduced network round trips. Table DDL Change Notification is also useful for OCI clients of the database that cache table metadata in the middle tier to avoid round trips to the database.

An OCI client can register to receive table DDL change notification for any of the following:

- A list of tables
- A list of schemas
- All tables in a database

Table DDL Change Notification is included in Oracle AI Database 26ai, and later releases.

35.2 Table DDL Change Notification Terminology

OCI Client

An OCI client (client, client application) is a client program that calls the APIs in Oracle Call Interface (OCI).

Copyright © 1999, 2025, Oracle and/or its affiliates.

EMON

Notification systems use the Event monitor (EMON) server pool to publish notifications to OCI clients.

Notification

A notification is a message describing the table DDL event that is sent to an OCI client. A notification does not contain the entire DDL statement but provides the table name and type of DDL operation.

Subscription

A subscription defines a channel that the clients can use to receive a particular notification.

Event

An event is a message published on a subscription, describing the DDL action on a table.

Registration

A registration represents a client that wants notification for a particular subscription topic.

35.3 Benefits of Table DDL Change Notification

Table DDL Change Notification works without affecting the application activity in the database.

The following are some of the benefits of using Table DDL Change Notification:

- Notifications can be enabled on highly active tables.
- Registering and event processing do not block the DDL activity on a table.
- Registrations and notifications are processed with minimal overhead on DML activity and without invalidating table cursors.
- Registrations are processed quickly while allowing concurrent registrations of other client applications.
- Client applications can dynamically subscribe to any number of additional tables or schemas.

35.4 Features of Table DDL Change Notification

The following are the features of Table DDL Change Notification:

- DDL events are processed asynchronously without blocking user DML queries.
- The client application can choose to include optional events, such as Partition Maintenance Operations (PMOP) or truncate, which are not available for event registration by default.
- DDL notifications are staged in the System Global Area (SGA). If an instance restarts, unprocessed notifications in SGA are lost. Persistent events are not supported.
- On failover of the database, any undelivered events in SGA are lost. The client caches must be invalidated on failover to avoid reconciliation issues in the table metadata due to lost notifications.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- For logical standby and physical standby, the OCI client must register again over the new primary database to resume notification.

35.5 Using Table DDL Change Notification

Here is how the notification process works with Table DDL Change Notification:

- A client application registers for receiving notification about the DDL events on a table as a part of a subscription.

  > **See Also**
  >
  > Registering for Table DDL Change Notification for more information about registrations.

- A DDL event is generated when any DDL transaction commits to the table.
- The OCI EMON processes the DDL event and notifies the event to the client application in a native OCI format (`OCI_DTYPE_DDL_EVENT`).

  See DDL Event Payload in the following section for the information contained in a DDL event.
- The client application receives the DDL event and invokes a user callback to process the event.

  See Registering Client Callback for DDL Notification in the following section for more information about the user callback process.

DDL Event Payload

The event payload describes the DDL event that is notified to the client application. The attributes of the event include:

- Operation type: `CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`, or `FLASHBACK`
- Object name: The base table name
- Object type: The object affected by the DDL transaction, for instance, partition, index, or table
- Database name
- SCN: The commit SCN of the DDL transaction (to order events within the database).
- UTC Modification time: Useful to correlate events across databases

Registering Client Callback for DDL Notification

- A client uses the following API to register a user callback and process events:

  `OCISubscriptionRegister(subhp)`: With the `OCI_SUBSCR_CQ_QOS_DDL_NTFN` DDL notification QoS and the `DBCHANGE` namespace.
- This starts a client thread to asynchronously invoke the user callback on an event.
- The subscription is assigned a unique (system generated) registration ID (`regid`) and subscription name `DDNF<regid>`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> OCISubscriptionRegister() for more information about the user callback API.

Use Case: GoldenGate Replicat

Here is a use case that has Oracle GoldenGate Replicat as the client:

Oracle GoldenGate Replicat applies source DML and DDL changes at the target database during logical replication. During an application upgrade, in many instances, the target table structure is modified prior to making the corresponding changes to the source database. In such cases, DDL replication cannot be used to synchronize and reconcile the metadata between the source and the target database.

For example; suppose a source table called `FA1.AR_SALES_TAX` is being replicated to the target database. Replicat loads the metadata of `FA1.AR_SALES_TAX` to reconstruct the SQL statement `INSERT into FA1.AR_SALES_TAX values(..)`. The relevant metadata includes information, such as column names, column types, and whether the table is compressed. Since the table metadata rarely changes, Replicat caches the data instead of querying the target database repeatedly. When the table metadata is first cached, Replicat registers for table DDL events of `FA1.AR_SALES_TAX` on the target database.

Take a scenario where the `FA1.AR_SALES_TAX` table is modified on the target database directly (and not on the source). For example, additional columns are added to the target table without being added to the source database. Replicat is notified with the table name and the operation (for instance, `ALTER`). When Replicat receives a new transaction to apply, it can refresh its stale metadata based on the DDL events received and can replicate inserts into the newly added columns successfully.

35.6 Registering for Table DDL Change Notification

A client can register for DDL change notifications at the table level or the schema level (for schema-wide table DDL events).

To register for notifications, you must fulfill the following conditions:

- You must be a non-`SYS` user.
- You must have `SELECT ANY TABLE` privileges.
- You must have `CHANGE NOTIFICATION` privileges.
- Set up a TCP or IPC listener for OCI DDL Notification client connections, and set the `local_listener` parameter of the ROOT container to this listener. The `local_listener` parameter can be set in the initialization parameter file (`init.ora`), or as follows:

```
ALTER SYSTEM SET LOCAL_LISTENER=listener_name;
ALTER SYSTEM REGISTER;
```

> **Note**
>
> If a user loses the required privileges, the user's table or schema-level registrations are implicitly unregistered.

Copyright © 1999, 2025, Oracle and/or its affiliates.

35.6.1 Table-level Registration

To register for notifications at the table level, use the `OCIDdlEventRegister()` function.

> **See Also**
>
> OCIDdlEventRegister() in *Oracle Call Interface Developer's Guide* for more information about the `OCIDdlEventRegister()` function and examples.

Here are some important points to note while registering for table-level notifications:

- You can register to receive notifications on any number of tables.
- You can dynamically add or remove tables from the existing subscription.
- Adding or removing tables from an existing subscription is an idempotent operation.
- A table is implicitly unregistered when the table is dropped or renamed.
- Only existing tables can be registered.

35.6.2 Schema-level Registration

To register for table DDL notifications at the schema level, use the `OCIDdlEventRegister()` function.

> **See Also**
>
> OCIDdlEventRegister() in *Oracle Call Interface Developer's Guide* for more information about the `OCIDdlEventRegister()` function and examples.

Here are some important points to note while registering for schema-level notifications:

- You can register to receive notifications on any number of schema names.
- You can dynamically add or remove schemas from the existing subscription.
- Adding or removing schemas from an existing subscription is an idempotent operation.
- A schema is implicitly unregistered when the schema is dropped.
- Only existing schemas can be registered.

35.7 Unregistering for Table DDL Change Notifications

A client can unregister previously registered tables or schema using the `OCIDdlEventUnregister()` function. The client can select the tables and schemas to remove from registration.

To unregister DDL notifications at the table or schema level, use the `OCIDdlEventUnregister()` function.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> OCIDdlEventUnregister() in *Oracle Call Interface Developer's Guide* for more information about the `OCIDdlEventUnregister()` function and examples.

In the following cases, a table is automatically unregistered:

- The table or schema is dropped.
- A client process exits without unregistering.
- The table is renamed.
- The user who registered for notification loses the required privileges (`SELECT ANY TABLE` or `CHANGE NOTIFICATION`).

When a client application exits without unregistering, its unprocessed notifications are dropped when notification delivery fails. If the OCI client restarts, it needs to re-register to resume notifications.

35.8 Supported DDL Events and Commands

In general, space management operations on the table, partition or index are not supported.

Supported Events

The following DDL events are included for DDL notifications:

- Events on user tables and global temporary tables
- Events on special tables, such as AQ queues, blockchain tables, and materialized views
- Events on DDL statements that affect dependent objects, such as indexes and constraints of the base table.

The following DDL events have conditional support for DDL notifications:

- DDL can include filters and the client can include optional events, such as PMOP or `TRUNCATE` operations, which are not delivered by default.
- For multi-table DDL statements, such as foreign key updates and online redefinition, only tables that are registered for notification generate events.

The following DDL events are excluded from DDL notifications:

- Events on tables in Oracle-maintained schema and private or temporary tables
- DDL statements on triggers and packages
- Schema create and drop events

Supported commands

At the table level, the following commands are supported:

- `CREATE TABLE`
- `DROP TABLE`
- `TRUNCATE TABLE`
- `RENAME TABLE`

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `ALTER TABLE`

At the PMOP level, the following commands are supported:

- `CREATE TABLE PARTITION [SUBPARTITION]`
- `ALTER TABLE operation [PARTITION | SUBPARTITION]`

  Operation = `[ ADD | DROP | MODIFY | EXCHANGE | MERGE | RENAME | SPLIT | TRUNCATE ]`

At the constraint level, the following commands are supported:

`ALTER TABLE operation CONSTRAINT`

Operation = `[ADD | MODIFY | RENAME | DROP]`

At the Flashback level, the following commands are supported: `FLASHBACK TABLE`

At the index level, the following commands are supported:

- `CREATE INDEX`
- `ALTER INDEX`
- `DROP INDEX`

Unsupported Commands

At the table level, the following commands are not supported:

- `COMMENT`
- `SHRINK SEGMENT`
- `MOVE`

At the PMOP level, `MOVE` operations are not supported.

At the index level, space management operations like `REBUILD`, `COALESCE` and `SPLIT PARTITION` are not supported.

35.9 Monitoring Table DDL Change Notification

The registration identifier (ID) uniquely identifies a subscription. A client can register on mutliple tables or schemas in the database using the same subscription (meaning, registration ID). The following views can be queried by the registration ID to monitor table DDL events received for a subscription.

- DBA_DDL_REGS for information on all table, schema or database-level registrations.
- DBA_SUBSCR_REGISTRATIONS for more information about subscriptions created in the database.
- `V$SUBSCR_REGISTRATION_STATS` (V_SUBSCR_REGISTRATION_STATS) for notification statistics and diagnostic information about a subscription.

Copyright © 1999, 2025, Oracle and/or its affiliates.

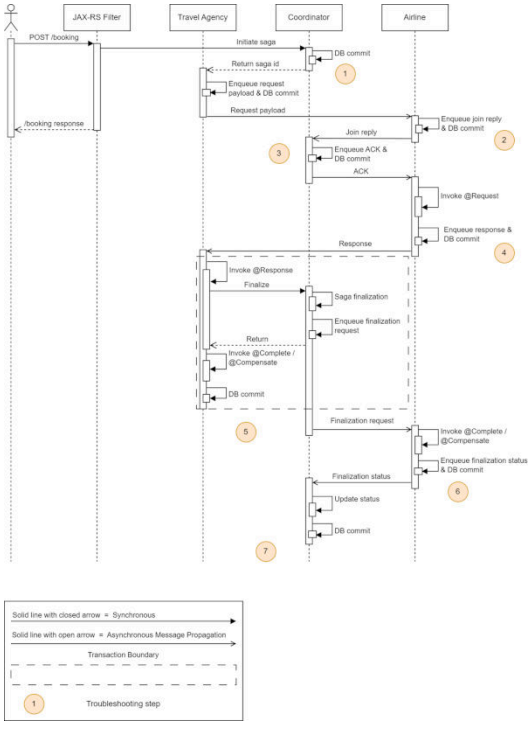
A.1 Appendix: Troubleshooting the Saga Framework

This appendix describes different ways of troubleshooting problems that may occur when you use the Saga framework with Java applications.

The Saga framework is a complex distributed system comprising multiple pluggable databases, and background and foreground database processes. The Saga framework uses database Advanced Queueing (AQ) or Transactional Event Queues (TxEventQ) as its event mesh. An event mesh seamlessly distributes events between various entities in a Saga topology. AQ message propagation and notification features enable event transmission to multiple entities in the Saga topology. The following call-return diagram depicts the lifetime of a simple Saga application that is described in the example program.

> **See Also**
>
> Example Program

Figure 1 Saga Lifecycle and Call-return Flow



Copyright © 1999, 2025, Oracle and/or its affiliates.

Note

- The Saga framework uses message propagation between the initiator and the participant using a message broker as an intermediary. It also relies on AQ notification framework to allow participants and coordinator to respond to the messages received. The message broker is not depicted in this picture.
- A database commit (DB commit) differs from a Saga commit (Saga finalize), in that, a database commit commits the local database transaction at the participant or initiator databases performed for a Saga. The Saga commit, on the other hand, commits the Saga and all its transactions performed by various participants on behalf of a Saga. The Saga commit is the final commit for all local transactional changes in a Saga as a group. These transactional changes were committed by various Saga participants previously using their respective DB commits. Conversely, a Saga rollback cancels the local transactional changes for a Saga as a group by doing compensation.
- The Saga framework provides an asynchronous platform for developing Sagas. The asynchronous platform is supported by AQ messaging in the database. Certain operations, however, are synchronous and accomplished inside the database server. For example, Saga initialization is a synchronous operation. Saga finalization at the coordinator and at the initiator is another example of a synchronous operation.
- For the current version of Saga infrastructure, the initiator and coordinator are co-located in a PDB. A co-located initiator and coordinator allows the finalization of a Saga using a single database transaction encompassing both entities. See the Transaction Boundary in the diagram highlighting this aspect.

A.1.1 Tracking a Saga

This section discusses about how you can keep track of the various states in a Saga at the initiator, participant and AQ topology levels.

A.1.1.1 Saga Initiator and Participant

Saga Initiator

On the initiator's PDB, the information about ongoing Sagas is available from the `DBA_SAGAS` and `DBA_SAGA_PARTICIPANT_SET` dictionary views. The `DBA_SAGAS` view provides the current state of ongoing Sagas, and `DBA_SAGA_PARTICIPANT_SET` shows the state of individual participants in a Saga. The completed Sagas appear in the `DBA_HIST_SAGAS` view.

Saga Participant

On the participant’s PDB, the information about ongoing Sagas is available from the `DBA_SAGAS` view. The completed Sagas appear in the `DBA_HIST_SAGAS` view.

A.1.1.2 AQ Topology

The following describes the AQ topology that supports the Saga framework. This information is available in the `DBA_SAGA_PARTICIPANTS` dictionary view. For the following example, let us assume that the broker named TestBroker is created on the Broker PDB.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Entity Name	Queue Name	Description
TestBroker	SAGA$_TESTBROKER_INOUT1	This queue represents the In/Out queue for a Saga broker. This queue is connected to other queues using AQ propagation.
TravelAgency	SAGA$_TRAVELAGENCY_IN_Q1	The initiator's IN queue stores incoming participant responses.
	SAGA$_TRAVELAGENCY_OUT_Q1	The initiator's OUT queue stores outgoing request messages.
TACoordinator	SAGA$_TACOORDINATOR_IN_Q1	For notification, use: "plsql://dbms_saga_adm.notify_callback_coordinator"
		The coordinator's IN queue stores incoming acknowledgements and finalization status messages.
	SAGA$_TACOORDINATOR_OUT_Q1	The coordinator's OUT queue stores outgoing acknowledgments and finalization requests.
Airline	SAGA$_AIRLINE_IN_Q1	The participant's IN queue stores the incoming payload requests and acknowledgments.
	SAGA$_AIRLINE_OUT_Q1	The participant's OUT queue stores the outbound messages bound to both initiators and coordinators.

A.1.1.3 Saga State Machine

A Saga transitions through the following states during its lifetime. For an example of this state transition, see the following section on "Troubleshooting Steps".

State	Description
Initiated	A new Saga is marked Initiated on the initiator PDB.
Joining	The Joining state in a Saga implies that the given participant is joining the Saga. This state is only relevant on the participant's PDB while the participant waits for an acknowledgment from the Saga coordinator to join the Saga.
Joined	A Saga is in the Joined state when it is successfully acknowledged by the Saga coordinator. This state is only relevant on the participant's PDB.
Finalization	A Saga undergoing commit or rollback has the Finalization state. Sagas undergoing finalization are listed in the DBA_HIST_SAGAS view.
Committed/Rolled back	A Saga that is committed or rolled back has this sate. Finalized Saga are listed in DBA_HIST_SAGAS view.
Auto Rolledback	A Saga is marked auto rolled back if it exceeds the Saga duration and is automatically rolled back by the initiator.

A.1.1.4 Saga Participant States

The DBA_SAGA_PARTICIPANT_SET view shows the state of a participant once it is associated with a Saga. The participant transitions through the following states:

Copyright © 1999, 2025, Oracle and/or its affiliates.

State	Description
Joined	The participant has `Joined` the Saga and is processing the request payload.
Committed/Rolledback	The Saga has been finalized by the participant.
Committed/Rollback Failed	The Saga finalization has failed.
Auto Rolledback	The Saga has been automatically rolled back as it exceeded its duration.
Rejected	The participant has rejected the join Saga request.

A.1.2 Troubleshooting Steps

1. Java code enters the `@LRA` annotated method `booking()` of the `TravelAgencyController` class.

   A query on `DBA_SAGAS` on the `TravelAgency's` PDB (`Travel PDB`) shows the following state:

   Travel PDB:

   ```
   SELECT id, initiator, coordinator, status FROM dba_sagas;
   ```

id	initiator	coordinator	status
abc123	TravelAgency	TACoordinator	Initiated

   At this point, no participants have been enrolled. The Saga identifier 'abc123' has been assigned for the newly created Saga.

2. This step corresponds to the participant `Airline` formally joining the Saga by sending an ACK to the Saga coordinator. This is handled by the Saga framework and is initiated by the join message from the Saga initiator. The Saga initiator invokes the `sendRequest()` call as shown in the following code segment:

   ```
   saga.sendRequest ("Airline", bookingPayload);
   ```

   The ACK is recorded at both the coordinator and participant PDBs. The following steps: 3 and 4, represent the chronological order of operations, as shown through the Saga dictionary views on the respective PDBs.

   Airline PDB (ACK initiated)

   ```
   SELECT id, participant, initiator, status FROM dba_sagas WHERE id='abc123';
   ```

id	participant	initiator	status
abc123	Airline	TravelAgency	Joining

3. The travel coordinator (`TACoordinator`) receives the asynchronous ACK message and responds by adding the participant (`Airline`) to the participant set for the given Saga and sending a message in response.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Travel Coordinator PDB (ACK received):

```
SELECT id, participant, status FROM dba_saga_participant_set WHERE
id='abc123';
```

id	participant	status
abc123	TravelAgency	Joined

4. The `Airline` receives the ACK message from the Saga coordinator and initiates the processing of the Saga payload using its `@Request` annotated method: `handleTravelAgencyRequest()`.

   Airline PDB (ACK complete):

   ```
   SELECT id, participant, initiator, status FROM dba_sagas WHERE id='abc123';
   ```

id	participant	initiator	status
abc123	Airline	TravelAgency	Joined

   The process of join acknowledgment is conducted using AQ messaging, propagation, and notification. The join message is enqueued at the source OUT queue (`SAGA$_AIRLINE_OUT_Q1` in our example) and propagated to the Saga coordinator's IN queue (`SAGA$_TACOORDINATOR_IN_Q1`) by the way of the message broker.

5. Upon receiving a response from `Airline`, the initiator (`TravelAgency`) finalizes (commits) the Saga. The finalization process involves the following state transitions.

   Travel PDB:

   ```
   SELECT id, initiator, coordinator, status FROM dba_sagas WHERE id='abc123';
   ```

id	initiator	coordinator	status
abc123	TravelAgency	TACoordinator	Committed

6. The `Airline` receives the commit message and initiates its own commit action. This results in the Saga transitioning from `Joined` to `Committed` on the participant PDB. The `Airline` sends a message to the coordinator indicating the status of its commit.

   Airline PDB:

   ```
   SELECT id, participant, initiator, status FROM dba_sagas WHERE id='abc123';
   ```

id	participant	initiator	status
abc123	Airline	TravelAgency	Committed

7. The final step corresponds to the `TACoordinator` registering the finalization status for `Airline`. This is reflected in the `dba_saga_participant_set` view.

   Travel Coordinator PDB:

   ```
   SELECT id, participant, status FROM dba_saga_participant_set WHERE
   id='abc123';
   ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

id	participant	status
abc123	TravelAgency	Committed

A.1.3 Diagnosing AQ Issues

As previously noted, the Saga framework uses asynchronous messaging (AQ) as its event mesh. The Saga framework relies on AQ message propagation and message notification. To diagnose issues with propagation and notification, see Managing Queues in the *Oracle Database Reference guide*.

Key elements to monitor are:

- Queue Depth for Saga queues (messages pending consumption)
- Propagation latency in the topology
- Notification latency

A.1.4 Saga Performance

The performance and throughput of Saga applications depend on the Saga framework and AQ performing at an optimal level. The database systems (PDBs) should be configured with adequate `JOB_QUEUE_PROCESSES` to facilitate AQ propagation and notification. Java applications should configure sufficient producers and consumers. See `numListeners` and `numPublishers` in the Configuration section of the Saga annotations.

In addition, the following parameters can be modified to achieve a higher level of performance.

`queue_partitions`

For higher throughput, Saga entities can deploy several queue partitions. By default, Saga entities use a single queue partition. Multiple queue partitions allow for a greater degree of parallelism for the Saga framework. All entities in the topology must have a matching number of queue partitions configured.

> **See Also**
>
> DBMS_SAGA_ADM for a complete description of the `SYS.DBMS_SAGA_ADM` package APIs.

`listener_count`

By default, the Saga framework uses the AQ message notification feature for handling the message traffic on the coordinator's IN queue. Under high load, the coordinator's IN queue may become a bottleneck for performance. The `listener_count` parameter of the `DBMS_SAGA_ADM.ADD_COORDINATOR()` procedure can be revised to a number greater than 1. This enables Saga message listeners to process messages on the Saga coordinator's IN queue instead of using AQ notifications. Saga message listeners eliminate some of the overheads of AQ message notification and provide an efficient mechanism for handling the inbound message traffic for the Saga coordinator.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> DBMS_SAGA_ADM for a complete description of the `SYS.DBMS_SAGA_ADM` package APIs.

A.1.5 Logging

The Saga framework supports the SLF4J logging facade, allowing end users to plug in logging frameworks of their own choice at deployment time. It is important, regardless of the logging framework, to ensure the proper permissions and best practices are applied to the log files.

Copyright © 1999, 2025, Oracle and/or its affiliates.

B.1 Appendix: Troubleshooting UTL_HTTP

This appendix guides you through the steps to troubleshoot issues that may arise while using the UTL_HTTP package.

The UTL_HTTP package is used to access a given URL from either an internal, external, or a secure website. Troubleshooting this package may require expertise from different competence areas. This guide takes you through a checklist of items. Depending on where an error occurs in the context of the checklist, this guide determines the competence area best suited for further assistance with the actual issue at hand.

UTL_HTTP Package Overview

You can use a UTL_HTTP package (PL/SQL package) method to obtain HTML text from a given web server page. The obtained text can be used within your application (usually by parsing the text for data). Your application would execute a PL/SQL procedure within the database to call the UTL_HTTP package after passing in the desired parameters. The derived result can be processed to perform a variety of tasks.

The UTL_HTTP package is used within the database. The package makes an HTTP or HTTPS connection on the internet or intranet, and brings back text, which an application can process. No HTTP server, except that which it is accessing, is required for this purpose. It can be any web server (not necessarily an Oracle server) that serves HTTP or HTTPS requests. When making HTTP callouts from PL/SQL or SQL, it turns the database into a text-based browser.

The UTL_HTTP package is not the same as the PL/SQL web toolkit, which is used with the Oracle HTTP Server to access procedures in the database and generate HTML pages to return back to the browser. While the UTL_HTTP package can be used within an application that is accessed over the web, its processing does not send anything back to the browser (unless at a further point in your application, the code calls the PL/SQL Toolkit OWA and HTP packages).

Troubleshooting Steps

The troubleshooting involves the following checks that are enumerated in the following steps. It is recommended that you perform these tasks in the given order, and also ensure that each step is completed before moving on to the next step.

- Database Check (Steps 1 to 3) wherein you verify that the UTL_HTTP package is valid and the required privileges are set correctly
- Secure Website Access (HTTPS) Check (Steps 4 and 5) wherein you verify if a non-secure website is being accessed, which includes the following: if Oracle Wallet is being used, verify the wallet location, check if Oracle has the permission to open the wallet, and verify if the wallet password is correct
- Configuration Check (Step 6) wherein you verify if the UTL-HTTP package is being used in conjunction with another Oracle product
- Language Check (Step 7) wherein you ascertain if the language-handling group should be involved to troubleshoot programmatic issues

Step 1: Verify that the UTL_HTTP Package is Valid

To verify if the UTL_HTTP package is valid, use the following command.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> The dependent objects may differ based on the database version.

```
SQL> column object_name format a15

SQL>SELECT object_name,
           object_type,
           status
    FROM dba_objects
      WHERE object_name IN
         (SELECT referenced_name
            FROM dba_dependencies
            WHERE name='UTL_HTTP')
      ORDER BY object_name, object_type;
```

An example of the output is as follows:

```
OBJECT_NAME     OBJECT_TYPE         STATUS
--------------- ------------------- -------
PLITBLM         PACKAGE             VALID
PLITBLM         SYNONYM             VALID
STANDARD        PACKAGE             VALID
STANDARD        PACKAGE BODY        VALID
UTL_HTTP        PACKAGE             VALID
UTL_HTTP        PACKAGE BODY        VALID
UTL_HTTP        SYNONYM             VALID
UTL_HTT_LIB     LIBRARY             VALID
UTL_RAW         PACKAGE             VALID
UTL_RAW         PACKAGE BODY        VALID
UTL_RAW         SYNONYM             VALID
```

Another example of the output is as follows:

```
OBJECT_NAME     OBJECT_TYPE         STATUS
--------------- ------------------- -------
STANDARD        PACKAGE             VALID
STANDARD        PACKAGE BODY        VALID
UTL_HTTP        PACKAGE             VALID
UTL_HTTP        PACKAGE BODY        VALID
UTL_HTTP        SYNONYM             VALID
```

If any of the objects are not valid, run the `htlrp.sql` script to validate them.

```
cd $ORACLE_HOME/rdbms/admin sqlplus

SQL> SELECT object_name FROM DBA_OBJECTS WHERE status = 'INVALID';
SQL> connect / as sysdba
SQL> @utlrp.sql
SQL> quit
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Step 2: Verify if the Required Privileges are Set Correctly

If the packages are all being returned with a status of `VALID`, then check the privileges.

To check the privileges, use the following commands:

```
SQL> column grantee format a10
SQL> column owner format a6
SQL> column table_name format a15
SQL> column grantor format a10
SQL> column privilege format a10
SQL> SELECT * FROM dba_tab_privs WHERE table_name='UTL_HTTP';
```

An example of the output is as follows:

```
GRANTEE    OWNER  TABLE_NAME      GRANTOR    PRIVILEGE  GRA HIE
---------- ------ --------------- ---------- ---------- --- ---
PUBLIC     SYS    UTL_HTTP        SYS        EXECUTE    NO  NO
```

Step 3: Check the Alert Logs (`alert.log`)

If no errors are returned from the previous step, check the `alert.log` file for additional information that is relevant to the time the `UTL_HTTP` package was executed. If you are a DBA (Database Administrator), you can query the `background_dump_dest` initialization parameter to find the location of the `alert.log` file.

```
SQL> column value format a40
SQL> column name format a30
SQL> SELECT name, value FROM v$parameter WHERE name='background_dump_dest';
```

DBA Database Checks

If any of the following is true, then any further troubleshooting should be within the purview of the database DBA group.

- Package(s) is invalid.
- Privileges are not granted.
- There are errors in the `alert.log` file.
- Running any `UTL_HTTP` function or procedure results in an `ORA-00600` or `ORA-03113` error.

Step 4: Check if it is a Secure Website Access (HTTPS Access)

If the target URL of the website contains `https` instead of `http`, then the website is a secure website.

> **Note**
>
> If a browser is available on the server, you should verify that the secure URL can be accessed. If you cannot access the URL on the browser, then the `utl_http` package cannot access it either. Also, ensure that a dialog box requesting client authentication does not appear (as this is not yet supported).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Just as with the browser, the UTL_HTTP package also supports HTTP over the Secured Socket Layer (SSL) protocol (also known as HTTPS), directly or through an HTTP proxy. For releases prior to Oracle AI Database Release 26ai, an Oracle Wallet is required to make an HTTPS request using the UTL_HTTP package (Non-HTTPS fetches do not require an Oracle Wallet). Starting Oracle AI Database 26ai, you can use the operating system's certificate store instead of an Oracle Wallet (provided the web service you are connecting to is "trusted" by the operating system).

About Oracle Wallet

Oracle Wallet contains the list of certificate authorities that the user of the UTL_HTTP package trusts. When a wallet is created, it is populated with a set of well-known certificate authorities as trust points. If the certificate authority that signs the certificate of the remote HTTPS web server is not among the trust points, then you should obtain the root certificate of that certificate authority, and install it as a trust point in the wallet using Oracle Wallet Manager.

Step 5: Verify the Oracle Wallet Location

When the UTL_HTTP package is executed in the Oracle Database server, the wallet must be accessible. To confirm the existence of the wallet and also the file permissions, navigate to the wallet directory using the system shell. The directory should have a file named `ewallet.p12` and the file permissions should be set with at least read permissions for the Oracle user.

For example, on Unix, you should see something similar to the following:

```
[sunsys]/etc/ORACLE/WALLETS/oracle> ls -al

drwxr-xr-x   2 oracle   dba          512 Jan 28 11:33 ./
drwxr-xr-x  10 oracle   dba          512 Jan 30 08:39 ../
-r--------   1 oracle   dba         8581 Jan 17 11:31 ewallet.p12
```

With the wallet configured, you can test the access to the secure website using the following SQL:

```
SELECT utl_http.request('', '', 'file:', '') FROM DUAL;
```

For example:

```
SELECT utl_http.request('https://www.xyz.com','proxy.<Domin Name>:<Port
Number>','file:/etc/ORACLE/WALLETS/oracle','welcome1') FROM DUAL;
```

Step 6: Verify if the UTL_HTTP package is Being Used in Conjunction with Another Oracle Product

> **Note**
>
> Using SQL*Plus or PL/SQL does not constitute the use of another product.

Verify if the UTL_HTTP package is being used in conjunction with another Oracle product or component (such as, Reports, Portal, Discoverer). If so, any Support Service Request should be transferred to the associated competence area. However, it is recommended to test a plain or simple `.html` page. If a simple page works with UTL_HTTP, but another component's pages

Copyright © 1999, 2025, Oracle and/or its affiliates.

do not work, the issue is within the component being used; perhaps the way the page is rendered back to the `UTL_HTTP` package.

The reason for transferring to the associated group is because, at this point, the `UTL_HTTP` package has been confirmed to be successfully installed, and a connection to an internal and external website can be made (along with a secure site, if applicable). For example, when used with Reports, the URL for the Reports server is generally used and since connection to other sites works fine, the issue may be related to the actual Reports Server and the URL.

Step 7: Check the Language

With the addition of new functions within the `UTL_HTTP` package, the chance of integrating within PL/SQL-specific functionality has increased, and the language group should be involved if the customer has programmatic issues. This falls under the guidelines of using the results from the `UTL_HTTP` package within expressions, control structures (`IF-THEN-ELSE`), or loops (`Simple, While, For`).

If you can successfully accomplish all the tasks discussed so far, but the issue remains unresolved, it could likely be because of custom application usage, or a bug with the package or interaction with the database. Ensure that you always apply the latest database patch set, because the `UTL_HTTP` package would then include any new patch updates.

Step 8: Contact Oracle Support

If you are still unable to troubleshoot the issue, you can contact Oracle Support for further assistance. Be prepared with a test case that can be used to try and reproduce the issue.

Copyright © 1999, 2025, Oracle and/or its affiliates.

C.1 Appendix: Recording DML Changes on the Tracked Table

DML Changes on a Tracked Table Row	Temporary `tcrv` Table	Blockchain History Table
First Insert		For any row, its very first inserted data (first lifespan) is protected by inserting a dummy row into its associated blockchain history table. This dummy row records the digest over the row's first lifespan.
Update	The latest version of the row (with the updates to the columns) now becomes the current lifespan of this tracked table row and gets recorded in the `tcrv` table. The beginning SCN of this current lifespan is taken from the commit SCN of the transaction that inserted the new row. The end SCN is infinite. The previous current lifespan is deleted from the `tcrv` table.	The previous current lifespan is inserted into the history table. The beginning SCN of the entry in the history table is the commit SCN of the first transaction that inserted the row, and its end SCN is the commit SCN of the second transaction that updated the row. Based on the current lifespan row in the `tcrv` table, a cryptographic digest (SHA2-512) is computed for the row with the latest lifespan and the cryptographic hash of the previous row in the chain. This digest is inserted in the blockchain history table in the column `ORAFBA_CURRENT_LIFESPAN_DIGEST$` along with the previous lifespan. Therefore, this new column protects the latest lifespan row in the `tcrv` table, while also protecting itself using the blockchain semantics.
Delete	The current lifespan in the `tcrv` table is removed.	A new historical lifespan is created in the history table (indicating that the `ROWID` is deleted).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Index

Numerics

32-bit IEEE 754 format, *8*
64-bit IEEE 754 format, *8*

A

Abstract Data Type (ADT), *12*, *8*
    native floating-point data types in, *12*
    resetting evolved, *8*
ACCESSIBLE BY clause, *2*
    in package specification, *2*
    stored subprogram and, *1*
accessor list
    See ACCESSIBLE BY clause
actual object, *11*
actualization, *11*
ADDM (Automatic Database Diagnostic Monitor), *8*
address of row (rowid), *24*
administrators, restricting with Oracle Database Vault, *2*
ADT
    See Abstract Data Type (ADT)
AFTER SUSPEND trigger, *43*
agent, *3*
aggregate function, *1*
ALL_ARGUMENTS, *16*
ALL_DEPENDENCIES, *16*
ALL_ERRORS, *16*
ALL_IDENTIFIERS, *16*
ALL_STATEMENTS, *16*
altering application online
    See edition-based redefinition (EBR)
analytic function, *8*
ancestor edition, *11*
annotations, *88*
    annotation DDL statements topics, *91*
    annotation syntax, *91*
    annotations and comments, *90*
    DDL statments
        domains, *96*
        indexes, *96*
        table columns, *93*
        tables, *92*
        views and materialized views, *94*
annotations *(continued)*
    dictionary table and views, *97*
        querying dictionary views, *98*
    overview, *89*
    privileges, *90*
    supported objects, *90*
ANSI data type, *24*
ANYDATA data type, *22*
ANYDATASET data type, *22*
AP (application program), *2*
application architecture, *2*
Application Continuity
    RESET_STATE, *7*
application domain index, *2*
application program (AP), *2*
application SQL, *30*
APPLYING_CROSSEDITION_TRIGGER function, *33*
AQ (Oracle Advanced Queuing), *2*
archive
    See Flashback Data Archive, *20*
ARG*n* data type, *26*
arithmetic operation
    with datetime data type, *17*
    with native floating-point data type, *11*
assignment, reported by PL/Scope, *6*
auditing
    available options, *6*
    unified auditing, *6*
auditing policy, editioning view and, *41*
AUTHID clause
    in package specification, *2*
    stored subprogram and, *1*
AUTHID property
    of invoked subprogram, *23*
    of PL/SQL unit, *3*, *30*
auto-tuning OCI client statement cache, *26*
Automatic Database Diagnostic Monitor (ADDM), *8*
Automatic Undo Management system, *1*
Automatic Workload Repository (AWR), *21*
autonomous transaction, *35*
    nonblocking DDL statement in, *35*
    trigger as, *43*

Copyright © 1999, 2025, Oracle and/or its affiliates.

B

backend for Microservices and AI, *1*
  about, *1*
  CloudBank sample application, *3*
backward compatibility
  LONG and LONG RAW data types for, *20*
  RESTRICT_REFERENCES pragma for, *35*
BATCH commit redo option, *6*
benchmark, *4*
binary floating-point number, *8*
binary format, *9*
binary large object (BLOB) data type, *19*
BINARY_DOUBLE data type, *7*
BINARY_FLOAT data type, *7*
BINARY_INTEGER data type
  See PLS_INTEGER data type
bind variables, *1*
block, PL/SQL, *1*
blocking DDL statement, *35*
BOOLEAN data type, *10*
branch, *2*
built-in data type
  See SQL data type
built-in function
  See SQL function
bulk binding, *18*
business rule, *1*

C

C external subprogram, *39*
  callback with, *39*
  global variable in, *43*
  interface between PL/SQL and, *11*
  invoking, *32*
  loading, *4*
  passing parameter to, *16*
  publishing, *13*
  running, *29*
  service routine and, *33*
    See also external subprogram
call specification
  for external subprogram, *3*
  in package, *2*
  location of, *13*
CALL statement, *29*
calling subprogram
  See invoking subprogram
cascading invalidation, *5*
CHANGE_DUPKEY_ERROR_INDEX hint, *33*
CHAR data type, *6*
character data type class, *16*
character data types, *6*
character large object (CLOB) data type, *19*
CHECK constraint
  compared to NOT NULL constraint, *17*
  designing, *17*
  multiple, *17*
  naming, *34*
  restrictions on, *16*
  when to use, *16*
client configuration parameter, *21*
client notification, *2*
client result cache, *11*
client statement cache auto-tuning (OCI client session feature), *26*
CLIENT_RESULT_CACHE_LAG server initialization parameter, *21*
CLIENT_RESULT_CACHE_SIZE server initialization parameter, *20*
client/server architecture, *2*
CLOB data type, *6*
coarse-grained invalidation, *5*
collection, *11*
  referenced by DML statement, *19*
  referenced by FOR loop, *20*
  referenced by SELECT statement, *19*
column
  generated, *5*
  multiple foreign key constraints on, *12*
  virtual, *5*
  when to use default value for, *6*
commit redo management, *6*
COMPATIBLE server initialization parameter, *20*
compilation parameter, *3*
composite PL/SQL data type, *11*
concurrency
  serializable transaction for, *27*
  under explicit locking, *19*
conditional compilation, *2*
connection class, *36*
connection pool, *34*
connection pools
  connection storms, *1*
  design guidelines for logins, *3*
  design guildelines, *1*
  drained, *4*
  guideline for preventing connection storms, *2*
  guidelines for preventing programmatic session leaks, *3*
  lock leaks, *4*
  logical corruption, *4*
constraint, *1*, *4*
  altering, *38*
  CHECK
    See CHECK constraint, *16*
  compared to trigger, *2*
  crossedition trigger and
    collisions, *33*
    dropping, *37*

Copyright © 1999, 2025, Oracle and/or its affiliates.

constraint *(continued)*
deferring checking of, *12*
disabling
effect of, *34*
existing, *36*
new, *35*
reasons for, *34*
dropping, *40*
editioning view and, *26*
enabling
effect of, *34*
existing, *36*
new, *35*
exception to, *37*
FOREIGN KEY
See FOREIGN KEY constraint, *9*
minimizing overhead of, *14*
naming, *34*
on view, *1*
PRIMARY KEY
See PRIMARY KEY constraint, *7*
privileges needed for defining, *33*
referential integrity
See FOREIGN KEY constraint, *9*
renaming, *39*
UNIQUE
See UNIQUE constraint, *8*
viewing definition of, *42*
violation of, *37*
Continuous Query Notification (CQN), *27*, *1*
example, *38*
converting data types
See data type conversion
copy-on-change strategy, *11*
coupling, *2*
CQ_NOTIFICATION$_DESCRIPTOR object, *48*
CQ_NOTIFICATION$_QUERY object, *50*
CQ_NOTIFICATION$_REG_INFO object, *21*
CQ_NOTIFICATION$_ROW object, *50*
CQ_NOTIFICATION$_TABLE object, *49*
CQN (Continuous Query Notification), *1*
CREATE OR REPLACE optimization, *5*
actualization and, *11*
CREATE_COVERAGE_TABLES procedure, *2*
cross-session PL/SQL function result cache, *17*
crossedition trigger, *27*
creating, *32*
displaying information about, *40*
dropping, *37*
execution of, *31*
forward, *28*
interaction with editions, *28*
read-only editioning view and, *25*
read-write editioning view and, *25*
reverse, *28*
scope of, *2*
crossedition trigger *(continued)*
sharing child cursor and, *40*
crossedition trigger SQL
forward, *29*
reverse, *30*
current date and time, displaying, *14*
current edition, *17*
cursor, *12*
crossedition trigger and, *40*
explicit, *12*
implicit, *12*
Oracle XA application and, *12*
schema object dependency and, *20*
session, *12*
cursor variable, *10*
advantages of, *13*
disadvantages of, *14*

D

data cartridge, *3*
data definition language statement
See DDL statement
data integrity, *1*
data type and, *2*
See also constraint
data modeling, *1*
data type
ANSI, *24*
BOOLEAN, *10*
DB2, *24*
dynamic, *22*
external, *1*
for character data, *6*
for datetime data, *13*
for geographic data, *19*
for large amount of data, *19*
for numeric data, *7*
for spatial data, *19*
for XML data, *21*
importance of correct, *1*
PL/SQL, *9*
PLS_INTEGER, *10*
REF CURSOR, *10*
SQL, *1*
SQL/DS, *24*
data type class, *16*
data type conversion, *12*
of ANSI and IBM data types, *24*
of datetime data types, *18*
of native floating-point data types, *12*
data type family
PL/SQL, *9*
SQL, *26*
database hardening, *20*
database logins, automated, *2*

Copyright © 1999, 2025, Oracle and/or its affiliates.

Database Resident Connection Pool (DRCP), *27*
date, *13*
  default format for, *14*
  displaying, *15*
    current, *14*
  inserting, *15*
    See also datetime data types
datetime data type class, *16*
datetime data types, *13*
  arithmetic operations with, *17*
  conversion functions for, *18*
  importing, exporting, and comparing, *19*
day, default value for, *16*
DB2 data type, *24*
DBA_STATEMENTS, *12*
DBA_STATEMENTS.SIGNATURE, *15*
DBA_STATEMENTS.TYPE Column, *12*
DBA_STATEMENTS.USAGE_CONTEXT_ID, *14*
DBA_STATEMENTS.USAGE_ID, *13*
DBMS_APPLICATION_INFO package, *5*
DBMS_DEBUG_JDWP, *39*
DBMS_DEBUG_JDWP package, *38*
DBMS_FLASHBACK package, *15*
  version query, *16*
DBMS_FLASHBACK_ARCHIVE procedures, *20*
DBMS_FLASHBACK_ARCHIVE_MIGRATE, *28*
DBMS_FLASHBACK.TRANSACTION_BACKOUT
    procedure, *17*
DBMS_HPROF package, *2*
DBMS_HPROF.ANALYZE, *21*
DBMS_LOCK package, *26*
DBMS_OUTPUT package, *38*
DBMS_PARALLEL_EXECUTE package, *35*
DBMS_PLSQL_CODE_COVERAGE, *1*, *2*
DBMS_SQL.RETURN_RESULT procedure, *16*
DBMS_STATS package, *38*
DBMS_TYPES package, *22*
DBMS_XA package, *17*
DDL statement, *2*
  blocking, *35*
  Flashback Archive, *22*
  for creating package, *7*
  for creating subprogram, *7*
  ineffective, *5*
  nonblocking, *35*
    in autonomous transaction, *35*
  Oracle XA and, *28*
  processing, *2*
  that generates notification, *6*
DDL_LOCK_TIMEOUTparameter, *35*
deadlock, undetected, *26*
debugging
  compiling PL/SQL unit for, *39*
  external subprogram, *42*
  subprogram, *38*
  wrap utility and, *39*
decimal floating-point number, *8*
default column value, *6*
deferring constraint checks, *12*
definer's rights, *4*
denormal floating-point number, *9*
dependency mode, *13*
dependent object
  See schema object dependency
dependent transaction, *18*
DEPRECATE PRAGMA, *22*
descendent edition, *11*
design
  physical, *2*, *3*
DETERMINISTIC function
  function-based index and, *4*
  RPC signature and, *14*
developing Java saga applications, *13*
developing Java Saga applications, *1*
  saga annotations
    example program, *35*
developing saga applications, *1*
  JMS Interface
    initialization, *19*
    JMS message properties, *7*
  PL/SQL interface
    example program, *12*
  saga framework
    aftersaga callback, *43*
    finalizing a saga, *39*
    lock-free reservation integration, *42*
    managing sagas using PL/SQL, *11*
    message propagation, *9*
    saga framework setup example, *10*
    saga interface, *21*
    SagaInitiatior class, *33*
    SagaMessageContext Class, *30*
    SagaParticipant class, *30*
  saga interface, *21*
  why, *6*
developing Saga applications
  implementing Sagas, *1*
  JMS Interface
    JMS message properties, *19*
  Oracle Saga framework
    overview, *2*
  saga annotations, *14*
  saga framework
    adding a coordinator, *7*
    adding a message broker, *7*
    adding a participant, *8*
    concepts, *3*
    dictionary tables, *9*
    features, *2*
    initializing, *6*
    managing participants and brokers, *8*
    PL/SQL callbacks, *40*

Copyright © 1999, 2025, Oracle and/or its affiliates.

developing Saga applications *(continued)*
saga framework *(continued)*
setting up Sagas, *6*
dirty read, *28*
disabling constraint
effect of, *34*
existing, *36*
new, *35*
reasons for, *34*
distributed database
FOREIGN KEY constraint and, *15*
remote dependency management and, *11*
distributed transaction, *2*
remote subprogram and, *29*
DLL (dynamic link library), *3*
DML statement
bulk binding for, *19*
that references collection, *19*
DML_LOCKS initialization parameter, *13*
domain index, *2*
double-precision IEEE 754 format, *8*
drivers, Oracle JDBC, *6*
DTP (X/Open Distributed Transaction
architecture), *1*
dynamic link library (DLL), *3*
dynamic registration, *2*
dynamic SQL, *1*, *21*
implicit query results and, *16*
RESTRICT_REFERENCES pragma and, *38*
dynamically typed data, *22*

E

EBR (edition-based redefinition), *1*
edition, *2*
ancestor, *11*
creating, *11*
crossedition triggers and, *28*
current, *17*
descendent, *11*
displaying information about, *38*
dropping, *22*
enabling for user and types, *6*
evaluation
See evaluation edition, *4*
leaf, *11*
making available
to all users, *16*
to some users, *16*
ora$base, *2*, *11*
retiring, *21*
root, *11*
scope of, *2*
session, *17*
unusable
See unusable edition, *4*
edition *(continued)*
visibility of trigger in, *28*
edition-based redefinition (EBR), *1*
EDITIONABLE property, *9*
editionable schema object type, *5*
editioned object, *2*
creating or replacing, *10*
name resolution and, *3*
editioning view, *24*
auditing policy and, *41*
changing base table of, *26*
changing writability of, *26*
covering table with, *41*
creating, *25*
displaying information about, *39*
partition-extended name for, *25*
preparing application for, *41*
read-only, *25*
read-write, *25*
replacing, *26*
scope of, *2*
SQL optimizer hint and, *27*
efficient table DDL change notification
overview, *1*
Electronic Product Code (EPC), *23*
embedded PL/SQL gateway, *3*
how to use, *4*
enabling constraint
effect of, *34*
existing, *36*
new, *35*
enabling editions, *6*
encoding scheme, adding, *13*
enumeration use case domains, *31*
about enumeration type, *31*
associating with existing table columns, *38*
associating with new table columns, *34*
creating, *32*
overview, *31*
using DML, *35*
environment, programming, *1*
EPC (Electronic Product Code), *23*
evaluation edition, *4*
dropping edition and, *23*
for materialized view, *4*
for virtual column, *5*
retiring edition and, *21*
exception, *2*
IEEE 754 standard
not raised, *10*
raised during conversion, *12*
in multilanguage program, *33*
to constraint, *37*
exception handling, *2*
for storage allocation error, *42*

Copyright © 1999, 2025, Oracle and/or its affiliates.

EXCLUSIVE MODE option of LOCK TABLE statement, *17*
EXECUTE privilege, *7*
execution plan, *7*
  data type and, *3*
EXPLAIN PLAN statement, *7*
explicit cursor, *12*
EXPR data type, *26*
expression
  index built on
    See function-based index, *2*
  regular
    See regular expression, *1*
extensibility, *2*
external data type, *1*
external large object (BFILE) data type, *19*
external subprogram, *3*, *12*, *39*
  call specification for, *3*
  debugging, *42*
  loading, *4*
  managing for applications, *6*
  publishing, *9*
external transaction manager, *2*

F

family of data types
  PL/SQL, *9*
  SQL, *26*
FAN event, load balancing advisory, *8*
Fast Application Notification (FAN), *5*
fine-grained access control, *2*
fine-grained auditing (FGA) policy, editioning view and, *41*
fine-grained invalidation, *5*
firing order of triggers, *30*
FIXED_DATE initialization parameter, *14*
fixed-point data type, *7*
Flashback Archive, *20*
Flashback Data Archive, *20*
Flashback Time Travel, *20*
Flashback Transaction, *17*
flexible use case domains
  associating domains with columns when creating tables, *26*
  associating domains with existing columns, *29*
  creating, *24*
  disassociating a domain from a column, *30*
  using DML, *27*
FLOAT data type, *7*
floating-point data type, *7*
  range and precision of, *8*
    See also native floating-point data type
floating-point number
  binary, *8*
  components of, *8*
floating-point number *(continued)*
  decimal, *8*
  denormal, *9*
  format of, *8*
  rounding, *8*
  subnormal, *9*
FOR loop
  bulk binding for, *20*
  that references collection, *20*
FORCE option of ALTER USER statement, *7*
FOREIGN KEY constraint, *9*
  distributed databases and, *15*
  dropping, *40*
  editioned view and, *10*
  enabling, *40*
  Flashback Transaction and, *18*
  indexing, *15*
  multiple, *12*
  naming, *34*
  NOT NULL constraint on, *11*
  NULL value and, *11*
  privileges needed to create, *41*
  referential integrity enforced by, *41*
  UNIQUE constraint on, *11*
  without other constraints, *11*
foreign key dependency, *18*
forward compatibility, *2*
forward crossedition trigger, *28*
forward crossedition trigger SQL, *29*
function, *1*
  aggregate, *1*
  analytic, *8*
  built-in
    See SQL function, *25*
  DETERMINISTIC
    function-based index and, *4*
    RPC signature and, *14*
  invoking from SQL statement, *30*
  MGD_ID ADT, *11*
  OCI or OCCI, *38*
  PARALLEL_ENABLE, RPC signature and, *14*
  PL/SQL, invoked by SQL statement, *30*
  purity of, *32*
    RPC signature and, *14*
  result-cached, *17*
  returning large amount of data from, *17*
  SQL
    See SQL function, *18*
  SQL analytic, *8*
    See also subprogram
function result cache, *17*
function-based index, *2*
  example for faster case-insensitive searches, *8*
  example for object columns, *7*

Copyright © 1999, 2025, Oracle and/or its affiliates.

function-based index *(continued)*
example for precomputing arithmetic expressions, *6*
optimizer and, *2*, *3*

G

generated column, *5*
Geographic Information System (GIS) data, *19*
GET_LTXID_OUTCOME procedure, *9*
global transaction, *2*
global variable, in C external subprogram, *43*
greedy operator in regular expression, *5*
group commit, *6*

H

hierarchical profiler, *1*
historical data, importing and exporting, *20*
host language, *33*
host program, *33*
hot rollover, *1*

I

IA-32 and IA-64 instruction set architecture, *11*
IBM CICS, *2*
IBM Transarc Encina, *2*
Identity Code Package, *1*
IEEE 754 standard, *7*
exception
not raised, *10*
raised during conversion, *12*
special values supported by, *10*
See also native floating-point data type
IGNORE_ROW_ON_DUPKEY_INDEX hint, *33*
IMMEDIATE commit redo option, *6*
implementing database application, *3*
implicit connection pooling, *39*
choose pool boundary, *43*
CMAN-TDM in PRCP, *43*
impact of OCI calls, *42*
resetting session states, *44*
security, *45*
session cached cursors, *45*
session states, *41*
statement and transaction boundary, *41*
implicit connection pooling)
configuring boundaries, *42*
POOL_BOUNDARY attribute, *42*
implicit cursor, *12*
in-flight transaction, *7*
independent transaction
See autonomous transaction
index, *1*
domain, *2*
index *(continued)*
edition-based redefinition and, *26*
function-based
See function-based index, *2*
on MGD_ID column, *10*
infinity, *10*
INHERIT ANY PRIVILEGES system privilege, *5*
INHERIT PRIVILEGES privilege, *5*
inherited object, *11*
initialization parameter, *3*
DML_LOCKS, *13*
FIXED_DATE, *14*
NLS_DATE_FORMAT, *14*
instrumentation, *2*
integer data type class, *16*
integrity constraint
See constraint
integrity of data
See data integrity
interface, *37*
between PL/SQL and C, *11*
between PL/SQL and Java, *11*
program, *3*
TX, *2*
user, *3*
stateful or stateless, *4*
See also Oracle C++ Call Interface
invalidation
cascading, *5*
coarse-grained, *5*
fine-grained, *5*
of dependent object, *5*
of package, *41*
invoker's rights, *4*
affect on invoker's privileges, *5*
Java stored procedures, *5*
invoking subprogram, *23*
from subprogram, *26*
from trigger, *26*
interactively from Oracle Database tools, *25*
through embedded PL/SQL gateway, *18*
isolation level
See transaction isolation level
iterative data processing
about, *3*
arrays, *4*
manual parallelism, *5*
row by row, *3*

J

Java class method, *12*
calling, *32*
interface between PL/SQL and, *11*
publishing, *12*
See also external subprogram

Copyright © 1999, 2025, Oracle and/or its affiliates.

Java Database Connectivity
See Oracle JDBC
Java language
compared to PL/SQL, *28*
Oracle Database support for, *4*
Java stored procedures, *5*
Java Virtual Machine
See Oracle JVM
JavaScript Support
overview, *13*
JDBC
See Oracle JDBC
JVM
See Oracle JVM

K

key
foreign
See FOREIGN KEY constraint, *9*
primary
See PRIMARY KEY constraint, *7*
unique
See UNIQUE constraint, *8*

L

Large Object (LOB), *19*
leaf edition, *11*
LGWR (log writer process), *6*
lightweight queue, *3*
live operation, *1*
load balancing advisory FAN event, *8*
LOB
See Large Object (LOB)
LOCK TABLE statement, *14*
SELECT FOR UPDATE statement with, *19*
lock-free reservation, *1*
about CHECK constraints, *8*
about concurrency in transactions, *1*
adding or modifying reservable columns, *7*
benefits, *11*
comparing optimistic and lock-free reservation, *5*
creating a reservable column, *5*
examples, *9*
guidelines, *11*
inserts and deletes guidelines, *13*
introduction, *3*
querying reservable column views, *10*
reservable column guidelines, *11*
reservation journal table columns, *6*
restrictions for reservation journal table, *14*
terminology, *2*
update statement guidelines, *12*
lock-less reservation
concurrent DDL guidelines, *13*
locking row explicitly, *18*
locking table
explicitly, *13*
implicitly, *17*
log writer process (LGWR), *6*
logical design, *2*
logical transaction identifier (LTXID), *8*
logical value, *10*
LONG and LONG RAW data types, *20*
LONG data type, *6*
loose coupling, *2*
LTXID (logical transaction identifier), *8*

M

main transaction, *35*
maintaining database and application, *4*
managing default rights, *5*
materialized view, *9*
that depends on editioned object, *4*
maximum availability of table, *25*
memory advisor, *9*
metacharacter in regular expression, *1*
metadata for SQL operator or function, *25*
metrics, *4*
MGD_ID ADT, *1*
MGD_ID database ADT function, *11*
microservice architecture, *1*
about, *1*
challenges, *3*
features, *3*
solutions, *4*
saga design pattern, *6*, *8*
implementation approaches, *7*
saga flow, *7*
two-phase commit, *5*
mod_plsql module, *2*
mode
dependency, *13*
lock, *14*
serialized
See serializable transaction, *27*
MODIFY CONSTRAINT clause of ALTER TABLE statement, *38*
modifying
See altering
monitoring database performance, *8*
multi-column use case domains
altering, *22*
creating, *18*
disassociating a domain from a column, *23*
dropping a domain, *23*, *30*
using DML, *20*

Copyright © 1999, 2025, Oracle and/or its affiliates.

multilanguage program, *1*
  error or exception in, *33*
multiline mode, *2*
multilingual data, *8*
multilingual engine, *13*
  built-in MLE modules, *20*
  call specification overview, *17*
  calling JavaScript functions, *20*
  concepts, *14*
  create MLE call specification clauses, *22*
  dynamic MLE execution overview, *23*
  invoking JavaScript, *17*
    using MLE modules, *18*
  invoking JavaScriptusing dynamic MLE execution, *22*
  JavaScript MLE modules overview, *16*
  managing JavaScript MLE modules, *19*
  MLE environment overview, *15*
  MLE execution contexts, *14*
  other supported MLE features, *27*
  privileges, *26*
  running JavaScript code inline, *25*
  running JavaScript Code using files, *25*
  runtime isolation for dynamic execution, *23*
  runtime isolation using MLE module contexts, *18*
  specifying an environment for a call specification, *19*
  specifying an environment on a call specification, *24*
  user privileges, *26*
  using dynamic MLE execution, *24*
  using JavaScript MLE modules, *20*

N

name resolution, *10*
  editions and, *3*
NaN (not a number), *10*
national character large object (NCLOB) data type, *19*
native execution, *23*
native floating-point data type, *7*
  arithmetic operation with, *11*
  binary format for, *9*
  conversion functions for, *12*
  in client interfaces, *12*
  special values for, *10*
NCHAR data type, *6*
NCLOB data type, *6*
negative infinity, *10*
negative zero, *10*
nested subprogram, *1*
NLS_DATE_FORMAT initialization parameter, *14*
NO_RESULT_CACHE hint, *14*
nonblocking DDL statement, *35*
  in autonomous transaction, *35*
NONEDITIONABLE property, *9*
noneditionable schema object type, *5*
noneditioned object, *2*
  creating or replacing, *10*
  name resolution and, *3*
  that can depend on editioned object, *4*
    dropping edition and, *23*
    FORCE and, *7*
nongreedy operator in regular expression, *9*
nonpersistent queue, *3*
normalized significand, *9*
NOT NULL
  See NOT NULL constraint, *4*
NOT NULL constraint
  compared to CHECK constraint, *17*
  naming, *34*
  on FOREIGN KEY constraint, *11*
  when to use, *4*
NOWAIT commit redo option, *6*
NOWAIT option of LOCK TABLE statement, *14*
NULL value
  FOREIGN KEY constraint and, *11*
  function-based index and, *2*
  indexing and, *4*
NUMBER data type, *7*
number data type class, *16*
numeric data types, *7*
NVARCHAR2 data type, *6*

O

object
  actual, *11*
  dependent
    See schema object dependency, *1*
  editioned
    See editioned object, *2*
  inherited, *11*
  large
    See Large Object (LOB), *19*
  noneditioned
    See noneditioned object, *2*
  potentially editioned, *2*
    with noneditioned dependents, *7*
  referenced
    See schema object dependency, *1*
  size limit for PL/SQL stored, *8*
object change notification, *2*
OCCI
  See Oracle C++ Call Interface
OCI
  See Oracle Call Interface
OCI_ATTR_CHDES_DBNAME, *36*
OCI_ATTR_CHDES_NFTYPE, *36*

Copyright © 1999, 2025, Oracle and/or its affiliates.

OCI_ATTR_CHDES_TABLE_CHANGES, *36*
OCI_ATTR_CHDES_TABLE_NAME, *37*
OCI_ATTR_CHDES_TABLE_OPFLAGS, *37*
OCI_ATTR_CHDES_TABLE_ROW_CHANGES, *37*
OCI_ATTR_CHDES_XID, *36*
OCI_ATTR_CHNF_CHANGELAG, *34*
OCI_ATTR_CHNF_ROWIDS, *34*
OCI_ATTR_CQ_QUERYID, *36*
OCI_ATTR_CQDES_OPERATION, *37*
OCI_ATTR_CQDES_QUERYID, *37*
OCI_ATTR_CQDES_TABLE_CHANGES, *37*
OCI_ATTR_SESSION_STATE attribute, *50*
OCI_ATTR_SUBSCR_CALLBACK, *34*
OCI_ATTR_SUBSCR_CQ_CHNF_QOSFLAGS, *34*
OCI_ATTR_SUBSCR_TIMEOUT, *34*
OCI_DTYPE_CQDES, *37*
OCI_SECURE_NOTIFICATION, *33*
OCI_SESSGET_PURITY_NEW attribute, *35*
OCI_SESSGET_PURITY_SELF attribute, *35*
OCI_SESSION_STATELESS attribute, *50*
OCI_SUBSCR_QOS_PURGE_ON_NTFN, *34*
OCIAnyData and OCIAnyDataSet interfaces, *22*
ODP.NET, *41*
online application upgrade
  See edition-based redefinition (EBR)
operator
  in regular expression, *5*
    greedy, *5*
    nongreedy, *9*
  metadata for, *25*
optimizer
  editioning view and, *27*
  function-based index and, *2*, *3*
  RPC signature and, *14*
ora$base edition, *2*, *11*
Oracle Advanced Queuing (AQ), *2*
Oracle backend for Microservices and AI
  about, *1*
Oracle C++ Call Interface, *37*
  building application with, *39*
  kinds of functions in, *38*
  native floating-point data types in, *12*
  procedural and nonprocedural elements of, *39*
Oracle Call Interface, *37*
  building application with, *39*
  commit redo action in, *6*
  compared to precompiler, *40*
  kinds of functions in, *38*
  native floating-point data types in, *12*
  procedural and nonprocedural elements of, *39*
  with Oracle XA, *14*
Oracle Data Provider for .NET, *41*
Oracle Data Redaction, *2*
Oracle data type
  See SQL data type
Oracle Database Tuning Pack, *9*
Oracle Database Vault, *2*
Oracle Extensibility Architecture framework, user-defined aggregate functions and, *3*
Oracle Extensibility Architecture, data cartridges and, *3*
Oracle Flashback Data Archive
  Oracle Virtual Private Database, *35*
Oracle Flashback Query, *8*
Oracle Flashback Technology, *1*
  application development features, *2*
  configuring database for, *4*
  database administration features, *4*
  performance guidelines for, *38*
Oracle Flashback Transaction Query, *12*
Oracle Flashback Version Query, *10*
Oracle JDBC, *6*
  compared to Oracle SQLJ, *11*
  native floating-point data types in, *12*
  sample program
    2.0, *8*
    pre-2.0, *8*
Oracle JDeveloper, Oracle SQLJ and, *10*
Oracle JVM, *5*
Oracle Label Security, *2*
Oracle Lock Management services, *26*
Oracle Real Application Clusters (Oracle RAC)
  client result cache and, *20*
  DRCP and, *51*
  load balancing advisory FAN events and, *8*
  Oracle XA and, *24*
  runtime connection load balancing and, *5*
Oracle sessionless transactions, *1*
  benefits, *12*
  capabilities, *4*
  committing a sessionless transaction, *33*, *36*
  draining sessionless transactions, *51*
  errors messages, *48*
  finalizing a sessionless transaction, *33*
  introduction, *1*
  lifecycle, *14*
  OCI example, *40*
  Oracle Coordinated Distributed Transaction interoperability, *49*
  overview, *3*
    example use cases, *8*
  restrictions, *50*
  resuming a suspended sessionless transaction, *29*
  rules and guidelines, *46*
  starting a new sessionless transaction, *18*
    running DML statements, *22*
  suspending a sessionless transaction, *24*
  using, *13*
    active sessionless transaction, *14*

Copyright © 1999, 2025, Oracle and/or its affiliates.

Oracle sessionless transactions *(continued)*
using *(continued)*
getting a GTRID, *22*
prerequisites, *16*
server round trips pre-call and post-call functions, *14*
setting a GTRID, *16*
timeout of suspended sessionless transaction, *27*
Oracle SQLJ, *9*
compared to Oracle JDBC, *11*
Oracle JDeveloper and, *10*
Oracle Text, *21*
Oracle Total Recall, *20*
Oracle Tuxedo, *2*
Oracle Virtual Private Database
Oracle Flashback Data Archive, *35*
Oracle Virtual Private Database (VPD), *2*
editioning view and, *41*
Oracle XA
Oracle RAC and, *24*
subprograms, *5*
when to use, *1*
out-of-space error, *42*
overloaded subprogram, *1*

P

package
creating, *7*
dropping, *22*
invalidation of, *41*
session state and, *8*
size limit for, *8*
synonym for, *28*
package body, *2*
package invalidation and, *41*
package specification, *2*
package subprogram, *1*
PARALLEL_ENABLE function
RPC signature and, *14*
parallelized SQL statement, *34*
parameter
compilation
See compilation parameter, *3*
initialization, *3*
partition-extended editioning view name, *25*
partitioning, *10*
performance, *1*
performance goals, *4*
performance testing, *10*
performance, data type and, *3*
persistent LOB instance, *19*
persistent queue, *3*
phantom read, *28*
physical design, *3*
PL/Scope, *38*, *1*
PL/Scope security model, *2*
PL/Scope tool, *1*
PL/SQL block, *1*
PL/SQL data type, *9*
PL/SQL function result cache, *17*
PL/SQL gateway, *2*
PL/SQL hierarchical profiler, *1*
PL/SQL Hierarchical Profiler, *38*
PL/SQL language, *4*
compared to Java, *28*
PL/SQL object
See PL/SQL unit
PL/SQL optimize level, *4*
PL/SQL optimizer level, *2*
PL/SQL unit, *3*, *5*
compiling for debugging, *39*
CREATE OR REPLACE and, *5*
PL/SQL Web Toolkit, *3*
PLS_INTEGER data type, *10*
plshprof utility, *15*
PLSQL_CODE_COVERAGE package, *2*
pool, connection, *34*
positive infinity, *10*
positive zero, *10*
POSIX standard for regular expressions
operators defined in, *5*
Oracle SQL and, *4*
Oracle SQL multilingual extensions to, *8*
Oracle SQL PERL-influenced extensions to, *9*
potentially editioned object, *2*
with noneditioned dependents, *7*
pre-validating JSON data, *18*
enabling precheck for a new table, *26*
enabling precheck for an exisitng table, *29*
guidelines for PRECHECK constraints, *31*
precheck syntax and definition, *19*
supported conditions for JSON schema validation, *20*
precompiler, *33*
compared to Oracle Call Interface, *40*
Oracle XA and, *12*
PRIMARY KEY constraint, *7*
dropping, *40*
Flashback Transaction and, *18*
naming, *34*
primary key dependency, *18*
privileges, *1*
for debugging subprogram, *40*
for defining constraint, *33*
for Oracle Flashback Technology, *7*
for running subprogram, *24*
granting secure application roles, *1*
grouped into roles, *1*
INHERIT ANY PRIVILEGES system privilege, *5*

Copyright © 1999, 2025, Oracle and/or its affiliates.

privileges *(continued)*
INHERIT PRIVILEGES privilege, *5*
revoked, object dependency and, *8*
Pro*C/C++ precompiler, *34*
native floating-point data types in, *12*
Pro*COBOL precompiler, *36*
procedure, *1*
See also subprogram
product code, *23*
profiler, *1*
program interface, *3*
programming environment, *1*
public information, required, *5*
publish-subscribe model, *1*
purity of function, *32*
RPC signature and, *14*

Q

quality-of-service flag, *21*
query
registering for Continuous Query Notification, *11*
returning results to client, *12*
implicitly, *16*
Query Result Change Notification (QRCN), *2*
query rewrite, *9*
queue, *3*

R

Radio Frequency Identification (RFID) technology, *22*
RAW data type, *20*
raw data type class, *16*
READ COMMITTED transaction isolation level
compared to SERIALIZABLE, *34*
in Oracle Database, *28*
transaction interactions with, *28*
read consistency, *13*
statement-level, *12*
transaction-level, *12*
locking tables explicitly for, *13*
read-only transaction for, *12*
read lock, *31*
READ UNCOMMITTED transaction isolation level
in Oracle Database, *28*
transaction interactions with, *28*
read-only editioning view, *25*
read-only transaction, *12*
read-write editioning view, *25*
record, *11*
redefinition, edition-based (EBR), *1*
redo information for transaction, *6*
redo management, *6*
REF CURSOR data type, *10*
referenced object
See schema object dependency
referential integrity
serializable transactions and, *31*
trigger for enforcing, *31*
referential integrity constraint
See FOREIGN KEY constraint
REGEXP_COUNT function, *2*
REGEXP_INSTR function, *2*
REGEXP_LIKE condition, *2*
REGEXP_REPLACE function, *2*
back reference operator in, *5*
REGEXP_SUBSTR function, *2*
registering application data usage, *1*
registration
dynamic, *2*
for Continuous Query Notification, *11*
in publish-subscribe model, *3*
static, *2*
regular expression, *1*
in Oracle SQL, *2*
in SQL statement, *10*
metacharacter in, *1*
POSIX standard and
See POSIX standard for regular expressions, *4*
Unicode and, *4*
remote dependency management, *11*
remote procedure call dependency management, *13*
remote subprogram, *27*
repeatable read, *12*
locking tables explicitly for, *13*
read-only transaction for, *12*
REPEATABLE READ transaction isolation level
in Oracle Database, *28*
transaction interactions with, *28*
required public information, *5*
RESET_STATE, *7*
resource manager (RM), *2*
RESTRICT_REFERENCES pragma
for backward compatibility, *35*
static and dynamic SQL and, *38*
result cache, *17*
RESULT_CACHE hint, *14*
RESULT_CACHE_MODE session parameter, *15*
resumable storage allocation, *42*
RETENTION GUARANTEE clause for undo tablespace, *5*
RETENTION option of ALTER TABLE statement, *7*
RETURN_RESULT procedure, *16*
reverse crossedition trigger, *28*
reverse crossedition trigger SQL, *30*
RFID (Radio Frequency Identification) technology, *22*

Copyright © 1999, 2025, Oracle and/or its affiliates.

RM (resource manager), *2*
roles, *1*
root edition, *11*
rounding floating-point numbers, *8*
row
  address of (rowid), *24*
  locking explicitly, *18*
ROW EXCLUSIVE MODE option of LOCK TABLE statement, *15*
ROW SHARE MODE option of LOCK TABLE statement, *15*
rowid, *24*
ROWID data type
  ROWID pseudocolumn and, *24*
ROWID pseudocolumn, *24*
  CQN and, *13*
    See also rowid
RPC dependency management, *13*
  RPC signature and, *14*
RR datetime format element, *14*
rule on queue, *3*
rules engine, *3*
runtime connection load balancing, *5*
runtime error
  See exception

S

scalability, *1*
scalar PL/SQL data type, *9*
schema object dependency, *1*
  in distributed database, *11*
  invalidation and, *5*
  on nonexistence of other objects, *10*
  revoked privileges and, *8*
  shared pool and, *20*
schema object type
  editionable, *5*
  enabling for editions, *6*
  noneditionable, *5*
searchable text, *21*
secure application roles, *1*
security, *1*
  auditing, *6*
  external procedures for applications, *6*
  invoker's rights and definer's rights, *4*
  Java stored procedures default rights, *5*
  logon triggers and, *2*
  Oracle Data Redaction, *2*
  Oracle Database Vault, *2*
  Oracle Label Security, *2*
  Oracle Virtual Private Database, *2*
  privilege use, *1*
  privileges of invoking user, *5*
  role use, *1*
  secure application roles, *1*
SELECT FOR UPDATE statement, *18*
  LOCK TABLE statement with, *19*
  referential integrity and
    inside trigger, *31*
    outside trigger, *31*
SELECT statement
  bulk binding for, *19*
  referencing collection with, *19*
  with AS OF clause, *8*
  with FOR UPDATE clause
    See SELECT FOR UPDATE statement, *18*
  with VERSIONS BETWEEN clause, *10*
semi-available table, *25*
serendipitous change, *35*
  data transformation collisions and, *33*
  identifying, *33*
serializable transaction, *28*
  for concurrency control, *27*
  interaction with, *28*
  referential integrity and, *31*
SERIALIZABLE transaction isolation level, *28*
  compared to READ COMMITTED, *34*
  in Oracle Database, *28*
  transaction interactions with, *28*
    See also serializable transaction
server-side programming, *2*
service routine, C external subprogram and, *33*
session cursor, *12*
session edition, *17*
session purity, *35*
session state, *8*
session variable, *25*
set based processing, *8*
SET CONSTRAINTS statement, *12*
SET TRANSACTION statement with READ ONLY option, *12*
SHARE MODE option of LOCK TABLE statement, *16*
SHARE ROW EXCLUSIVE MODE option of LOCK TABLE statement, *17*
shared SQL area, *3*
side effects of subprogram, *32*
signature checking, *12*
single-column use case domains
  creating, *4*
single-precision IEEE 754 format, *8*
spatial data, *19*
specification, package
  See package specification
SQL Access Advisor, *9*
SQL advisor, *9*
SQL analytic function, *8*
SQL area, shared, *3*
SQL data type, *1*

Copyright © 1999, 2025, Oracle and/or its affiliates.

SQL function, *25*
analytic, *8*
display type of, *26*
for data type conversion, *18*
metadata for, *25*
SQL optimizer hint and editioning view, *27*
SQL statement, *30*
application, *30*
crossedition trigger
forward, *29*
reverse, *30*
invoking PL/SQL function from, *30*
parallelized, *34*
processing
DDL statement, *2*
stages of, *1*
system management statement, *2*
SQL Trace facility (SQL_TRACE), *6*
SQL Tuning Advisor, *9*
SQL, dynamic
See dynamic SQL
SQL/DS data type, *24*
SQLJ
See Oracle SQLJ
standalone subprogram, *1*
state
session, *8*
user interface and, *4*
web application and, *26*
stateful session, *50*
stateless session, *50*
statement
See SQL statement
statement caching, *25*
statement-level read consistency, *12*, *13*
static pools
used to prevent connection storms, *2*
static registration, *2*
static SQL, RESTRICT_REFERENCES pragma
and, *38*
static variable, in C external subprogram, *43*
statistics
for application, *1*
for identifier, *1*
storage allocation error, *42*
storage requirements, decreasing, *2*
stored standalone subprogram, dropping, *22*
stored subprogram, *1*
subnormal floating-point number, *9*
subprogram, *1*
creating, *7*
external
See external subprogram, *1*
invoking
See invoking subprogram, *23*
Oracle XA, *5*
subprogram *(continued)*
overloaded, *1*
privileges needed to debug, *40*
privileges needed to run, *24*
remote, *27*
size limit for, *8*
synonym for, *28*
subscriber, *3*
subscription services, *3*
subtype, *9*
user-defined, *11*
synonym
CREATE OR REPLACE and, *5*
for package, *28*
for subprogram, *28*
SYSDATE function, *14*
system management statement, *2*

T

table
controlling user access to, *2*
locking
choosing strategy for, *14*
explicitly, *13*
implicitly, *17*
with maximum availability, *25*
with semi-availability, *25*
table annotation, *15*
table DDL change notification, *1*
benefits, *2*
features, *2*
monitoring, *7*
registering, *4*
schema-level registration, *5*
supported DDL events, *6*
table-level registration, *5*
terminology, *1*
using, *3*
table result cache mode
annotation, *15*
effective
determining, *16*
displaying, *16*
Tag Data Translation Markup Language Schema,
1
Temporal Validity Support, *11*
temporary LOB instance, *19*
thin client configuration, *3*
thread-safe application, *17*
three-tier architecture, *3*
tight coupling, *2*
time, *13*
default value for, *16*
displaying, *16*
current, *14*

Copyright © 1999, 2025, Oracle and/or its affiliates.

time *(continued)*
inserting, *16*
See also datetime data types
time-stamp checking, *12*
time-stamp dependency mode, *13*
TM (transaction manager), *2*
TPM (transaction processing monitor), *2*
tracing tools, *1*
transaction
autonomous, *35*
trigger as, *43*
choosing isolation level for, *34*
dependent, *18*
determining outcome after outage, *7*
distributed, *2*
remote subprogram and, *29*
ensuring idempotence of, *7*
global, *2*
grouping operations into, *4*
improving performance of, *5*
in-flight, *7*
main, *35*
read-only, *12*
redo entry for, *6*
serializable
See serializable transaction, *27*
that invokes remote subprogram, *29*
transaction guard
rolling operation support, *14*
transient logical standby, *14*
Transaction Guard, *7*, *1*
transaction history table, *7*
transaction interaction
kinds of, *28*
serializable transaction and, *28*
transaction isolation level and, *28*
transaction isolation level, *28*
choosing, *34*
setting, *30*
transaction interaction and, *28*
transaction manager (TM), *2*
transaction processing monitor (TPM), *2*
transaction set consistency, *33*
transaction-level read consistency, *12*
locking tables explicitly for, *13*
read-only transaction for, *12*
transform, *28*
applying, *35*
Transparent Application Continuity
RESET_STATE, *7*
trigger, *1*
AFTER SUSPEND, *43*
as autonomous transaction, *43*
automating database login with, *2*
compared to constraint, *2*
trigger *(continued)*
crossedition
See crossedition trigger, *27*
enforcing referential integrity with, *31*
in edition
firing order of, *30*
visibility of, *28*
what kind can fire, *29*
invoking subprogram from, *26*
size limit for, *8*
TRUST assertion (deprecated), *37*
TRUST keyword in RESTRICT_REFERENCES pragma, *37*
two-phase commit protocol, *2*
two-tier architecture, *3*
TX interface, *2*

U

undetected deadlock, *26*
undo data, *1*
UNDO_RETENTION parameter, *12*
Unicode
data types for, *6*
regular expressions and, *4*
unified auditing, *6*
UNIQUE constraint
crossedition trigger and, *33*
dropping, *40*
naming, *34*
on FOREIGN KEY constraint, *11*
when to use, *8*
unrepeatable read, *28*
unusable edition
dropping edition and, *23*
for materialized view, *4*
retiring edition and, *21*
upgrading applications online
See edition-based redefinition (EBR)
UROWID data type, *24*
use case domains, *1*
altering, *12*
built-in domains, *49*
creating
specifying data type, *38*
dictionary views, *48*
dropping, *16*
drop domain and recycle bin, *17*
overview, *2*
SQL functions, *3*, *47*
use case domain types, *2*
Using
associating domains with columns when creating tables, *6*
associating domains with existing columns, *10*

Copyright © 1999, 2025, Oracle and/or its affiliates.

use case domains *(continued)*
Using *(continued)*
changing the domain associated with a column, *43*
disassociating a domain from a column, *14*
using DML, *8*
Using multi-column domains
associating domains with columns when creating tables, *19*
associating domains with existing columns, *21*
user access
See security
user interface, *3*
stateful and stateless, *4*
user lock, *26*
user-defined subtype, *11*
using IF EXISTS and IF NOT EXISTS, *45*
ALTER command, *46*
CREATE command, *45*
CREATE OR REPLACE limitations, *48*
DROP, *46*
SQL Plus DDL output messages, *48*
supported object types, *47*
UTLLOCKT.SQL script, *27*

V

VARCHAR data type, *6*
VARCHAR data type class, *16*
VARCHAR2 data type, *6*
variable
cursor
See cursor variable, *10*, *12*
in C external subprogram
global, *43*
variable *(continued)*
in C external subprogram *(continued)*
static, *43*
view
constraint on, *1*
editioned, FOREIGN KEY constraint and, *10*
editioning
See editioning view, *24*
materialized, *9*
that depends on editioned object, *4*
virtual column, *5*
VPD policy, editioning view and, *41*

W

WAIT commit redo option, *6*
WAIT option of LOCK TABLE statement, *14*
web application, *1*
implementing, *2*
state and, *26*
web services, *12*
web toolkit
See PL/SQL Web Toolkit
wrap utility, debugging and, *39*
writability of editioning view, *26*
write-after-write dependency, *18*

X

X/Open Distributed Transaction Processing (DTP) architecture, *1*
xa_open string, *8*
XMLType data type, *21*

Y

YY datetime format element, *15*

Copyright © 1999, 2025, Oracle and/or its affiliates.