# Oracle® Database

## Database Concepts

21c
F31733-07
April 2025

Oracle Database Database Concepts, 21c

F31733-07

Copyright © 1993, 2025, Oracle and/or its affiliates.

Primary Authors: Mark Doran, Lance Ashdown, Donna Keesling, Tom Kyte

Contributors: Drew Adams, Ashish Agrawal, Troy Anthony, Vikas Arora, Jagan Athraya, David Austin, Thomas Baby, Vladimir Barriere, Hermann Baer, Srinagesh Battula, Nigel Bayliss, Tammy Bednar, Virginia Beecher, Bjorn Bolltoft, Ashmita Bose, David Brower, Larry Carpenter, Lakshminaray Chidambaran, Deba Chatterjee, Shasank Chavan, Tim Chien, Gregg Christman, Bernard Clouse, Maria Colgan, Carol Colrain, Nelson Corcoran, Michael Coulter, Jonathan Creighton, Judith D'Addieco, Mark Dilman, Kurt Engeleiter, Bjørn Engsig, Marcus Fallon, Steve Fogel, Jonathan Giloni, Naveen Gopal, Bill Habeck , Min-Hank Ho, Lijie Heng, Bill Hodak, Yong Hu, Pat Huey, Praveen Kumar Tupati Jaganath, Sanket Jain, Prakash Jashnani, Caroline Johnston, Shantanu Joshi, Jesse Kamp, Vikram Kapoor, Feroz Khan, Jonathan Klein, Andre Kruglikov, Sachin Kulkarni, Surinder Kumar, Paul Lane, Adam Lee, Allison Lee, Jaebock Lee, Sue Lee, Teck Hua Lee, Yunrui Li , Ilya Listvinski, Bryn Llewellyn, Rich Long, Barb Lundhild, Neil Macnaughton, Vineet Marwah, Susan Mavris, Bob McGuirk, Joseph Meeks, Mughees Minhas, Sheila Moore, Valarie Moore, Gopal Mulagund, Charles Murray, Kevin Neel, Sue Pelski, Raymond Pfau, Gregory Pongracz, Vivek Raja, Ashish Ray, Bert Rich, Kathy Rich, Andy Rivenes, Scott Rotondo, Vivian Schupmann, Venkat Senaptai, Shrikanth Shankar, Prashanth Shanthaveerappa, Cathy Shea, Susan Shepard, Kam Shergill, Mike Skarpelos, Sachin Sonawane, James Spiller, Suresh Sridharan, Jim Stenoish, Janet Stern, Rich Strohm, Roy Swonger, Kamal Tbeileh, Juan Tellez, Ravi Thammaiah, Lawrence To, Tomohiro Ueda, Randy Urbano, Badhri Varanasi, Nick Wagner, Steve Wertheimer, Patrick Wheeler, Doug Williams, James Williams, Andrew Witkowski, Daniel Wong, Hailing Yu

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 xix
Documentation Accessibility xix
Related Documentation xx
Conventions xx

## 1 Introduction to Oracle Database

About Relational Databases 1-1
- Database Management System (DBMS) 1-2
- Relational Model 1-2
- Relational Database Management System (RDBMS) 1-3
- Brief History of Oracle Database 1-3

Schema Objects 1-5
- Tables 1-5
- Indexes 1-6

Data Access 1-6
- Structured Query Language (SQL) 1-6
- PL/SQL and Java 1-7

Transaction Management 1-8
- Transactions 1-8
- Data Concurrency 1-9
- Data Consistency 1-9

Oracle Database Architecture 1-10
- Database and Instance 1-10
  - Multitenant Architecture 1-10
  - Sharding Architecture 1-14
- Database Storage Structures 1-16
  - Physical Storage Structures 1-16
  - Logical Storage Structures 1-17
- Database Instance Structures 1-17
  - Oracle Database Processes 1-17
  - Instance Memory Structures 1-18
- Application and Networking Architecture 1-18

Application Architecture 1-19
Oracle Net Services Architecture 1-19
Oracle Database Documentation Roadmap 1-20
Oracle Database Documentation: Basic Group 1-21
Oracle Database Documentation: Intermediate Group 1-21
Oracle Database Documentation: Advanced Group 1-21

Part I Multitenant Architecture

2 CDBs and PDBs

About Containers in a CDB 2-1
The CDB Root and System Container 2-4
PDBs 2-5
Types of PDBs 2-6
Purpose of PDBs 2-7
Proxy PDBs 2-7
Names for PDBs 2-8
Database Links Between PDBs 2-9

3 Application Containers

About Application Containers 3-1
Purpose of Application Containers 3-2
Key Benefits of Application Containers 3-2
Application Container Use Case: SaaS 3-3
Application Containers Use Case: Logical Data Warehouse 3-4
Application Root 3-5
Application PDBs 3-6
Application Seed 3-6
Application Common Objects 3-6
About Commonality in a CDB 3-7
Principles of Commonality 3-7
Namespaces in a CDB 3-7
Creation of Application Common Objects 3-9
Metadata-Linked Application Common Objects 3-9
Metadata Links 3-11
Data-Linked Application Common Objects 3-11
Extended Data-Linked Application Objects 3-12
Container Maps 3-13

Cross-Container Operations 3-16

Part II Oracle Relational Data Structures

4 Tables and Table Clusters

Introduction to Schema Objects 4-2
About Common and Local User Accounts 4-2
Common User Accounts 4-4
Local User Accounts 4-8
Common and Local Objects 4-10
Schema Object Types 4-10
Schema Object Storage 4-11
Schema Object Dependencies 4-12
Sample Schemas 4-14
Overview of Tables 4-15
Columns 4-16
Virtual Columns 4-17
Invisible Columns 4-17
Rows 4-18
Example: CREATE TABLE and ALTER TABLE Statements 4-18
Oracle Data Types 4-20
Character Data Types 4-21
Numeric Data Types 4-23
Datetime Data Types 4-24
Rowid Data Types 4-25
Format Models and Data Types 4-27
Integrity Constraints 4-27
Table Storage 4-28
Table Organization 4-28
Row Storage 4-29
Rowids of Row Pieces 4-30
Storage of Null Values 4-30
Table Compression 4-30
Basic Table Compression and Advanced Row Compression 4-31
Hybrid Columnar Compression 4-32
Overview of Table Clusters 4-35
Overview of Indexed Clusters 4-36
Overview of Hash Clusters 4-39
Hash Cluster Creation 4-40
Hash Cluster Queries 4-40

Hash Cluster Variations 4-41
Hash Cluster Storage 4-42
Overview of Attribute-Clustered Tables 4-43
Advantages of Attribute-Clustered Tables 4-44
Join Attribute Clustered Tables 4-45
I/O Reduction Using Zones 4-45
Purpose of Zones 4-46
Zone Maps 4-46
Zone Map Creation 4-46
How a Zone Map Works: Example 4-48
Attribute-Clustered Tables with Linear Ordering 4-49
Attribute-Clustered Tables with Interleaved Ordering 4-50
Overview of Temporary Tables 4-51
Purpose of Temporary Tables 4-52
Segment Allocation in Temporary Tables 4-53
Temporary Table Creation 4-53
Overview of External Tables 4-53
Purpose of External Tables 4-54
Data in Object Stores 4-55
External Table Access Drivers 4-55
External Table Creation 4-56
Overview of Blockchain Tables 4-56
Row Chains 4-57
Row Content 4-57
User Interface for Blockchain Tables 4-58
Overview of Immutable Tables 4-59
Overview of Object Tables 4-59

## 5 Indexes and Index-Organized Tables

Introduction to Indexes 5-1
Advantages and Disadvantages of Indexes 5-3
Index Usability and Visibility 5-4
Keys and Columns 5-4
Composite Indexes 5-5
Unique and Nonunique Indexes 5-6
Types of Indexes 5-6
How the Database Maintains Indexes 5-8
Index Storage 5-8
Overview of B-Tree Indexes 5-8
Branch Blocks and Leaf Blocks 5-10
Index Scans 5-10

Full Index Scan 5-11
Fast Full Index Scan 5-12
Index Range Scan 5-12
Index Unique Scan 5-13
Index Skip Scan 5-14
Index Clustering Factor 5-15
Reverse Key Indexes 5-17
Ascending and Descending Indexes 5-17
Index Compression 5-18
Prefix Compression 5-18
Advanced Index Compression 5-20
Overview of Bitmap Indexes 5-21
Example: Bitmap Indexes on a Single Table 5-22
Bitmap Join Indexes 5-24
Bitmap Storage Structure 5-26
Overview of Function-Based Indexes 5-27
Uses of Function-Based Indexes 5-27
Optimization with Function-Based Indexes 5-28
Overview of Application Domain Indexes 5-29
Overview of Index-Organized Tables 5-30
Index-Organized Table Characteristics 5-31
Index-Organized Tables with Row Overflow Area 5-33
Secondary Indexes on Index-Organized Tables 5-34
Logical Rowids and Physical Guesses 5-35
Bitmap Indexes on Index-Organized Tables 5-36

## 6 Partitions, Views, and Other Schema Objects

Overview of Partitions 6-2
Partition Characteristics 6-3
Partition Key 6-3
Partitioning Strategies 6-3
Partitioned Tables 6-10
Segments for Partitioned Tables 6-11
Compression for Partitioned Tables 6-11
Partitioned Indexes 6-12
Local Partitioned Indexes 6-13
Global Partitioned Indexes 6-15
Partial Indexes for Partitioned Tables 6-18
Overview of Sharded Tables 6-19
Sharded Tables 6-20
Overview of Views 6-22

Characteristics of Views 6-23
Data Manipulation in Views 6-24
How Data Is Accessed in Views 6-24
Updatable Join Views 6-25
Object Views 6-26
Overview of Materialized Views 6-27
Characteristics of Materialized Views 6-29
Refresh Methods for Materialized Views 6-30
Complete Refresh 6-30
Incremental Refresh 6-30
In-Place and Out-of-Place Refresh 6-31
Automatic Materialized Views 6-31
Query Rewrite 6-32
Overview of Sequences 6-33
Sequence Characteristics 6-33
Concurrent Access to Sequences 6-33
Overview of Dimensions 6-34
Hierarchical Structure of a Dimension 6-35
Creation of Dimensions 6-35
Overview of Synonyms 6-36

## 7 Data Integrity

Introduction to Data Integrity 7-1
Techniques for Guaranteeing Data Integrity 7-1
Advantages of Integrity Constraints 7-2
Types of Integrity Constraints 7-3
NOT NULL Integrity Constraints 7-4
Unique Constraints 7-5
Primary Key Constraints 7-6
Foreign Key Constraints 7-7
Self-Referential Integrity Constraints 7-10
Nulls and Foreign Keys 7-10
Parent Key Modifications and Foreign Keys 7-11
Indexes and Foreign Keys 7-12
Check Constraints 7-12
States of Integrity Constraints 7-13
Checks for Modified and Existing Data 7-13
When the Database Checks Constraints for Validity 7-14
Nondeferrable Constraints 7-15
Deferrable Constraints 7-15
Examples of Constraint Checking 7-16

Example: Insertion of a Value in a Foreign Key Column When No Parent Key Value Exists 7-16
Example: Update of All Foreign Key and Parent Key Values 7-17

## 8 Data Dictionary and Dynamic Performance Views

Overview of the Data Dictionary 8-1
Purpose of the Data Dictionary 8-1
Data Management 8-2
Data Dictionary Separation in a CDB 8-2
Data Dictionary Components 8-3
Container Data Objects in a CDB 8-5
Views with the Prefix DBA_ 8-7
Views with the Prefix ALL_ 8-8
Views with the Prefix USER_ 8-8
The DUAL Table 8-9
How the Data Dictionary Works 8-9
Metadata and Data Links 8-10
Public Synonyms for Data Dictionary Views 8-11
Data Dictionary Cache 8-12
Other Programs and the Data Dictionary 8-12
Data Dictionary Storage 8-12
Overview of the Dynamic Performance Views 8-12
Contents of the Dynamic Performance Views 8-13
Storage of the Dynamic Performance Views 8-14
Database Object Metadata 8-14

# Part III Oracle Data Access

## 9 SQL

Introduction to SQL 9-1
SQL Data Access 9-2
SQL Standards 9-3
Overview of SQL Statements 9-3
Data Definition Language (DDL) Statements 9-4
Data Manipulation Language (DML) Statements 9-6
SELECT Statements 9-7
Joins 9-7
Subqueries 9-9
Transaction Control Statements 9-10
Session Control Statements 9-11

System Control Statement 9-11
Embedded SQL Statements 9-12
Overview of the Optimizer 9-12
Use of the Optimizer 9-13
Optimizer Components 9-14
Query Transformer 9-16
Estimator 9-16
Plan Generator 9-16
Access Paths 9-17
Optimizer Statistics 9-18
Optimizer Hints 9-19
Overview of SQL Processing 9-20
Stages of SQL Processing 9-20
SQL Parsing 9-21
SQL Optimization 9-22
SQL Row Source Generation 9-22
SQL Execution 9-23
Differences Between DML and DDL Processing 9-23

## 10 Server-Side Programming: PL/SQL and Java

Introduction to Server-Side Programming 10-1
Overview of PL/SQL 10-3
PL/SQL Subprograms 10-3
Advantages of PL/SQL Subprograms 10-4
Creation of PL/SQL Subprograms 10-5
Execution of PL/SQL Subprograms 10-6
PL/SQL Packages 10-7
Advantages of PL/SQL Packages 10-8
Creation of PL/SQL Packages 10-9
Execution of PL/SQL Package Subprograms 10-9
PL/SQL Anonymous Blocks 10-10
PL/SQL Language Constructs 10-11
PL/SQL Collections and Records 10-12
Collections 10-12
Records 10-13
How PL/SQL Runs 10-13
Overview of Java in Oracle Database 10-14
Overview of the Java Virtual Machine (JVM) 10-16
Overview of Oracle JVM 10-16
Main Components of Oracle JVM 10-17
Java Programming Environment 10-18

Java Stored Procedures 10-19
Java and PL/SQL Integration 10-19
Overview of Triggers 10-21
Advantages of Triggers 10-22
Types of Triggers 10-22
Timing for Triggers 10-23
Creation of Triggers 10-24
Example: CREATE TRIGGER Statement 10-25
Example: Invoking a Row-Level Trigger 10-26
Execution of Triggers 10-28
Storage of Triggers 10-28

## Part IV Oracle Transaction Management

## 11 Data Concurrency and Consistency

Introduction to Data Concurrency and Consistency 11-1
Multiversion Read Consistency 11-2
Statement-Level Read Consistency 11-3
Transaction-Level Read Consistency 11-4
Read Consistency and Undo Segments 11-4
Read Consistency and Deferred Inserts 11-6
Locking Mechanisms 11-7
ANSI/ISO Transaction Isolation Levels 11-7
Overview of Oracle Database Transaction Isolation Levels 11-8
Read Committed Isolation Level 11-9
Read Consistency in the Read Committed Isolation Level 11-9
Conflicting Writes in Read Committed Transactions 11-10
Serializable Isolation Level 11-12
Read-Only Isolation Level 11-16
Overview of the Oracle Database Locking Mechanism 11-17
Summary of Locking Behavior 11-17
Use of Locks 11-18
Lock Modes 11-21
Lock Conversion and Escalation 11-22
Lock Duration 11-22
Locks and Deadlocks 11-23
Overview of Automatic Locks 11-24
DML Locks 11-25
Row Locks (TX) 11-26
Table Locks (TM) 11-29

Locks and Foreign Keys 11-30
DDL Locks 11-34
Exclusive DDL Locks 11-34
Share DDL Locks 11-34
Breakable Parse Locks 11-35
System Locks 11-35
Latches 11-35
Mutexes 11-36
Internal Locks 11-36
Overview of Manual Data Locks 11-37
Overview of User-Defined Locks 11-38

## 12 Transactions

Introduction to Transactions 12-1
Sample Transaction: Account Debit and Credit 12-2
Structure of a Transaction 12-3
Beginning of a Transaction 12-4
End of a Transaction 12-4
Statement-Level Atomicity 12-6
System Change Numbers (SCNs) 12-6
Overview of Transaction Control 12-7
Transaction Names 12-9
Active Transactions 12-10
Savepoints 12-10
Rollback to Savepoint 12-11
Enqueued Transactions 12-11
Rollback of Transactions 12-13
Commits of Transactions 12-13
Overview of Transaction Guard 12-15
Benefits of Transaction Guard 12-16
How Transaction Guard Works 12-16
Lost Commit Messages 12-16
Logical Transaction ID 12-17
Transaction Guard: Example 12-18
Overview of Application Continuity 12-19
Benefits of Application Continuity 12-19
Use Case for Application Continuity 12-19
Application Continuity for Planned Maintenance 12-20
Application Continuity Architecture 12-21
Overview of Autonomous Transactions 12-22
Overview of Distributed Transactions 12-23

Two-Phase Commit 12-24
In-Doubt Transactions 12-24

Part V Oracle Database Storage Structures

13 Physical Storage Structures

Introduction to Physical Storage Structures 13-1
Mechanisms for Storing Database Files 13-3
Oracle Managed Files and User-Managed Files 13-4
Oracle Automatic Storage Management (Oracle ASM) 13-4
Oracle ASM Storage Components 13-5
Oracle ASM Instances 13-7
Oracle Persistent Memory Filestore (PMEM Filestore) 13-8
Directly Mapped Buffer Cache 13-8
File System Interface for PMEM Filestore 13-9
User Interface for PMEM Filestore 13-10
Overview of Data Files 13-10
Use of Data Files 13-10
Permanent and Temporary Data Files 13-12
Online and Offline Data Files 13-12
Data File Structure 13-13
Overview of Control Files 13-14
Use of Control Files 13-14
Multiple Control Files 13-15
Control File Structure 13-16
Overview of the Online Redo Log 13-17
Use of the Online Redo Log 13-17
How Oracle Database Writes to the Online Redo Log 13-17
Online Redo Log Switches 13-18
Multiple Copies of Online Redo Log Files 13-20
Archived Redo Log Files 13-21
Structure of the Online Redo Log 13-21

14 Logical Storage Structures

Introduction to Logical Storage Structures 14-1
Logical Storage Hierarchy 14-2
Logical Space Management 14-4
Locally Managed Tablespaces 14-5
Dictionary-Managed Tablespaces 14-8
Overview of Data Blocks 14-8

Data Blocks and Operating System Blocks 14-8
Database Block Size 14-9
Tablespace Block Size 14-10
Data Block Format 14-10
Data Block Overhead 14-11
Row Format 14-12
Data Block Compression 14-15
Space Management in Data Blocks 14-16
Percentage of Free Space in Data Blocks 14-16
Optimization of Free Space in Data Blocks 14-17
Chained and Migrated Rows 14-20
Overview of Index Blocks 14-22
Types of Index Blocks 14-22
Storage of Index Entries 14-23
Reuse of Slots in an Index Block 14-23
Coalescing an Index Block 14-23
Overview of Extents 14-25
Allocation of Extents 14-26
Deallocation of Extents 14-27
Storage Parameters for Extents 14-28
Overview of Segments 14-29
User Segments 14-30
User Segment Creation 14-30
Temporary Segments 14-32
Allocation of Temporary Segments for Queries 14-32
Allocation of Segments for Temporary Tables and Indexes 14-33
Undo Segments 14-33
Undo Segments and Transactions 14-35
Transaction Rollback 14-37
Temporary Undo Segments 14-37
Segment Space and the High Water Mark 14-38
Overview of Tablespaces 14-41
Tablespaces in a Multitenant Environment 14-41
Permanent Tablespaces 14-44
The SYSTEM Tablespace 14-45
The SYSAUX Tablespace 14-45
Undo Tablespaces 14-46
Shadow Tablespaces 14-47
Temporary Tablespaces 14-51
Shared and Local Temporary Tablespaces 14-51
Default Temporary Tablespaces 14-52
Tablespace Modes 14-54

Read/Write and Read-Only Tablespaces 14-55
Online and Offline Tablespaces 14-55
Tablespace File Size 14-56

# Part VI Oracle Instance Architecture

## 15 Oracle Database Instance

Introduction to the Oracle Database Instance 15-1
Database Instance Structure 15-2
Database Instance Configurations 15-3
Read/Write and Read-Only Instances 15-4
Duration of a Database Instance 15-5
Identification of a Database Instance 15-6
Oracle Base Directory 15-7
Oracle Home Directory 15-7
Oracle System Identifier (SID) 15-8
Overview of Database Instance Startup and Shutdown 15-9
Overview of Instance and Database Startup 15-9
Connection with Administrator Privileges 15-10
How an Instance Is Started 15-11
How a Database Is Mounted 15-12
How a Database Is Opened 15-12
Overview of Database and Instance Shutdown 15-14
Shutdown Modes 15-15
How a Database Is Closed 15-16
How a Database Is Unmounted 15-17
How an Instance Is Shut Down 15-17
Overview of Checkpoints 15-18
Purpose of Checkpoints 15-18
When Oracle Database Initiates Checkpoints 15-18
Overview of Instance Recovery 15-19
Purpose of Instance Recovery 15-20
When Oracle Database Performs Instance Recovery 15-20
Importance of Checkpoints for Instance Recovery 15-21
Instance Recovery Phases 15-22
Overview of Parameter Files 15-23
Initialization Parameters 15-23
Functional Groups of Initialization Parameters 15-23
Basic and Advanced Initialization Parameters 15-24
Server Parameter Files 15-24

Text Initialization Parameter Files 15-25
Modification of Initialization Parameter Values 15-26
Overview of Diagnostic Files 15-27
Automatic Diagnostic Repository 15-28
Problems and Incidents 15-29
ADR Structure 15-29
Alert Log 15-31
Attention Log 15-31
DDL Log 15-32
Trace Files 15-32
Types of Trace Files 15-32
Locations of Trace Files 15-33
Segmentation of Trace Files 15-33
Diagnostic Dumps 15-34
Trace Dumps and Incidents 15-34

## 16 Memory Architecture

Introduction to Oracle Database Memory Structures 16-1
Basic Memory Structures 16-2
Oracle Database Memory Management 16-3
Overview of the User Global Area 16-4
Overview of the Program Global Area (PGA) 16-5
Contents of the PGA 16-6
Private SQL Area 16-6
SQL Work Areas 16-8
PGA Usage in Dedicated and Shared Server Modes 16-9
Overview of the System Global Area (SGA) 16-9
Database Buffer Cache 16-11
Purpose of the Database Buffer Cache 16-11
Buffer States 16-12
Buffer Modes 16-13
Buffer I/O 16-13
Buffer Pools 16-17
Buffers and Full Table Scans 16-19
DRAM and PMEM Buffers 16-22
Redo Log Buffer 16-23
Shared Pool 16-24
Library Cache 16-25
Data Dictionary Cache 16-27
Server Result Cache 16-28
Reserved Pool 16-31

Large Pool 16-31
Large Pool Memory Management 16-32
Large Pool Buffers for Deferred Inserts 16-32
Java Pool 16-34
Fixed SGA 16-35
Optional Performance-Related SGA Subareas 16-35
In-Memory Area 16-35
Memoptimize Pool 16-36
Overview of Software Code Areas 16-37

## 17 Process Architecture

Introduction to Processes 17-1
Types of Processes 17-2
Multiprocess and Multithreaded Oracle Database Systems 17-4
Overview of Client Processes 17-5
Client and Server Processes 17-5
Connections and Sessions 17-6
Current Container 17-6
Database Operations 17-8
Overview of Server Processes 17-9
Dedicated Server Processes 17-10
Shared Server Processes 17-10
How Oracle Database Creates Server Processes 17-11
Overview of Background Processes 17-11
Mandatory Background Processes 17-12
Process Monitor Process (PMON) Group 17-13
Process Manager (PMAN) 17-15
Listener Registration Process (LREG) 17-15
System Monitor Process (SMON) 17-16
Database Writer Process (DBW) 17-16
Log Writer Process (LGWR) 17-17
Checkpoint Process (CKPT) 17-18
Manageability Monitor Processes (MMON and MMNL) 17-19
Recoverer Process (RECO) 17-19
Optional Background Processes 17-20
Archiver Processes (ARCn) 17-21
Job Queue Processes (CJQ0 and Jnnn) 17-21
Flashback Data Archive Process (FBDA) 17-22
Space Management Coordinator Process (SMCO) 17-22
Slave Processes 17-22
I/O Slave Processes 17-23

Parallel Execution (PX) Server Processes 17-24

## 18 Application and Oracle Net Services Architecture

Overview of Oracle Application Architecture 18-1
Overview of Client/Server Architecture 18-2
Distributed Processing 18-2
Advantages of a Client/Server Architecture 18-3
Overview of Multitier Architecture 18-4
Clients 18-5
Application Servers 18-5
Database Servers 18-5
Service-Oriented Architecture (SOA) 18-6
Overview of Grid Architecture 18-6
Overview of Oracle Net Services Architecture 18-7
How Oracle Net Services Works 18-8
The Oracle Net Listener 18-8
Service Names 18-10
Services in a Multitenant Environment 18-11
Service Registration 18-15
Dedicated Server Architecture 18-15
Shared Server Architecture 18-17
Dispatcher Request and Response Queues 18-18
Restricted Operations of the Shared Server 18-21
Database Resident Connection Pooling 18-21
Overview of the Program Interface 18-22
Program Interface Structure 18-23
Program Interface Drivers 18-23
Communications Software for the Operating System 18-24

## Glossary

## Index

# Preface

This manual provides an architectural and conceptual overview of the Oracle database server, which is an object-relational database management system.

The book describes how the Oracle database server functions, and it lays a conceptual foundation for much of the practical information contained in other manuals. Information in this manual applies to the Oracle database server running on all operating systems.

This preface contains these topics:


- Audience
- Documentation Accessibility
- Related Documentation
- Conventions



- Audience
- Documentation Accessibility
- Related Documentation
- Conventions


## Audience

*Oracle Database Concepts* is intended for technical users, primarily database administrators and database application developers, who are new to Oracle Database. Typically, the reader of this manual has had experience managing or developing applications for other relational databases.

To use this manual, you must know the following:

- Relational database concepts in general
- Concepts and terminology in Introduction to Oracle Database
- The operating system environment under which you are running Oracle

## Documentation Accessibility

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

**Access to Oracle Support**

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

# Related Documentation

For related documentation, see "Oracle Database Documentation Roadmap".

Many publications in the Oracle Database documentation set use the sample schemas of the database that is installed by default when you install Oracle Database. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them.

# Conventions

The following text conventions are used in this manual:

| 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 manual 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. |

1

# Introduction to Oracle Database

This chapter provides an overview of Oracle Database.

This chapter contains the following topics:


- About Relational Databases
- Schema Objects
- Data Access
- Transaction Management
- Oracle Database Architecture
- Oracle Database Documentation Roadmap


- About Relational Databases
  Every organization has information that it must store and manage to meet its requirements. For example, a corporation must collect and maintain human resources records for its employees. This information must be available to those who need it.
- Schema Objects
  One characteristic of an RDBMS is the independence of physical data storage from logical data structures.
- Data Access
  A general requirement for a DBMS is to adhere to accepted industry standards for a data access language.
- Transaction Management
  Oracle Database is designed as a multiuser database. The database must ensure that multiple users can work concurrently without corrupting one another's data.
- Oracle Database Architecture
  A **database server** is the key to information management.
- Oracle Database Documentation Roadmap
  The documentation set is designed with specific access paths to ensure that users are able to find the information they need as efficiently as possible.

## About Relational Databases

Every organization has information that it must store and manage to meet its requirements. For example, a corporation must collect and maintain human resources records for its employees. This information must be available to those who need it.

An information system is a formal system for storing and processing information. An information system could be a set of cardboard boxes containing manila folders along with rules for how to store and retrieve the folders. However, most companies today use a database to automate their information systems. A database is an organized collection of information treated as a unit. The purpose of a database is to collect, store, and retrieve related information for use by database applications.

- Database Management System (DBMS)
  A **database management system (DBMS)** is software that controls the storage, organization, and retrieval of data.
- Relational Model
  In his seminal 1970 paper "A Relational Model of Data for Large Shared Data Banks," E. F. Codd defined a relational model based on mathematical set theory. Today, the most widely accepted database model is the relational model.
- Relational Database Management System (RDBMS)
  The relational model is the basis for a **relational database management system (RDBMS)**. An RDBMS moves data into a database, stores the data, and retrieves it so that applications can manipulate it.
- Brief History of Oracle Database
  The current version of Oracle Database is the result of over 40 years of innovative development.

## Database Management System (DBMS)

A **database management system (DBMS)** is software that controls the storage, organization, and retrieval of data.

Typically, a DBMS has the following elements:

- Kernel code

  This code manages memory and storage for the DBMS.
- Repository of metadata

  This repository is usually called a data dictionary.
- Query language

  This language enables applications to access the data.

A database application is a software program that interacts with a database to access and manipulate data.

The first generation of database management systems included the following types:

- Hierarchical

  A hierarchical database organizes data in a tree structure. Each parent record has one or more child records, similar to the structure of a file system.
- Network

  A network database is similar to a hierarchical database, except records have a many-to-many rather than a one-to-many relationship.

The preceding database management systems stored data in rigid, predetermined relationships. Because no data definition language existed, changing the structure of the data was difficult. Also, these systems lacked a simple query language, which hindered application development.

## Relational Model

In his seminal 1970 paper "A Relational Model of Data for Large Shared Data Banks," E. F. Codd defined a relational model based on mathematical set theory. Today, the most widely accepted database model is the relational model.

A relational database is a database that conforms to the relational model. The relational model has the following major aspects:

- Structures

  Well-defined objects store or access the data of a database.

- Operations

  Clearly defined actions enable applications to manipulate the data and structures of a database.

- Integrity rules

  Integrity rules govern operations on the data and structures of a database.

A relational database stores data in a set of simple relations. A relation is a set of tuples. A tuple is an unordered set of attribute values.

A table is a two-dimensional representation of a relation in the form of rows (tuples) and columns (attributes). Each row in a table has the same set of columns. A relational database is a database that stores data in relations (tables). For example, a relational database could store information about company employees in an employee table, a department table, and a salary table.

**Related Topics**

- A Relational Model of Data for Large Shared Data Banks by E.F. Codd

## Relational Database Management System (RDBMS)

The relational model is the basis for a **relational database management system (RDBMS)**. An RDBMS moves data into a database, stores the data, and retrieves it so that applications can manipulate it.

An RDBMS distinguishes between the following types of operations:

- Logical operations

  In this case, an application specifies *what* content is required. For example, an application requests an employee name or adds an employee record to a table.

- Physical operations

  In this case, the RDBMS determines *how* things should be done and carries out the operation. For example, after an application queries a table, the database may use an index to find the requested rows, read the data into memory, and perform many other steps before returning a result to the user. The RDBMS stores and retrieves data so that physical operations are transparent to database applications.

Oracle Database is an RDBMS. An RDBMS that implements object-oriented features such as user-defined types, inheritance, and polymorphism is called an object-relational database management system (ORDBMS). Oracle Database has extended the relational model to an object-relational model, making it possible to store complex business models in a relational database.

## Brief History of Oracle Database

The current version of Oracle Database is the result of over 40 years of innovative development.

Highlights in the evolution of Oracle Database include the following:

- Founding of Oracle Corporation

  In 1977, Larry Ellison, Bob Miner, and Ed Oates started the consultancy Software Development Laboratories, which became Relational Software, Inc. (RSI). In 1983, RSI became Oracle Systems Corporation and then later Oracle Corporation.

- First commercially available RDBMS

  In 1979, RSI introduced Oracle V2 (Version 2) as the first commercially available SQL-based RDBMS, a landmark event in the history of relational databases.

- Portable version of Oracle Database

  Oracle Version 3, released in 1983, was the first relational database to run on mainframes, minicomputers, and personal computers. The database was written in C, enabling the database to be ported to multiple platforms.

- Enhancements to concurrency control, data distribution, and scalability

  Version 4 introduced multiversion read consistency. Version 5, released in 1985, supported client/server computing and distributed database systems. Version 6 brought enhancements to disk I/O, row locking, scalability, and backup and recovery. Also, Version 6 introduced the first version of the PL/SQL language, a proprietary procedural extension to SQL.

- PL/SQL stored program units

  Oracle7, released in 1992, introduced PL/SQL stored procedures and triggers.

- Objects and partitioning

  Oracle8 was released in 1997 as the object-relational database, supporting many new data types. Additionally, Oracle8 supported partitioning of large tables.

- Internet computing

  Oracle8i Database, released in 1999, provided native support for internet protocols and server-side support for Java. Oracle8i was designed for internet computing, enabling the database to be deployed in a multitier environment.

- Oracle Real Application Clusters (Oracle RAC)

  Oracle9i Database introduced Oracle RAC in 2001, enabling multiple instances to access a single database simultaneously. Additionally, Oracle XML Database (Oracle XML DB) introduced the ability to store and query XML.

- Grid computing

  Oracle Database 10g introduced grid computing in 2003. This release enabled organizations to virtualize computing resources by building a grid infrastructure based on low-cost commodity servers. A key goal was to make the database self-managing and self-tuning. Oracle Automatic Storage Management (Oracle ASM) helped achieve this goal by virtualizing and simplifying database storage management.

- Manageability, diagnosability, and availability

  Oracle Database 11g, released in 2007, introduced a host of new features that enabled administrators and developers to adapt quickly to changing business requirements. The key to adaptability is simplifying the information infrastructure by consolidating information and using automation wherever possible.

- Plugging In to the Cloud

  Oracle Database 12c, released in 2013, was designed for the Cloud, featuring a new Multitenant architecture, In-Memory Column Store (IM column store), and support for JSON documents. Oracle Database 12c helped DBAs make more efficient use of their IT resources, while continuing to reduce costs and improve service levels for end users.

- Integration and memory performance

  Oracle Database 18c simplified integration with directory services such as Microsoft Active Directory. It also introduced functionality to exploit memory for columnar data models and high-speed row access.

- Enhanced stability

  Oracle Database 19c was the long-support version of the Oracle Database 12c (Release 12.2) family of products. A major focus of this release was stability. Oracle Database 19c also introduced several small but significant improvements to features such as JSON and Active Data Guard.

- Improved developer experience

  Oracle Database 21c improves the developer experience with features such as Oracle Blockchain Tables and native JSON data types. Enhancements to Automatic In-Memory make the IM column store largely self-managing.

# Schema Objects

One characteristic of an RDBMS is the independence of physical data storage from logical data structures.

In Oracle Database, a database schema is a collection of logical data structures, or schema objects. A database user owns a database schema, which has the same name as the user name.

Schema objects are user-created structures that directly refer to the data in the database. The database supports many types of schema objects, the most important of which are tables and indexes.

A schema object is one type of database object. Some database objects, such as profiles and roles, do not reside in schemas.

- Tables
  A table describes an entity such as employees.
- Indexes
  An **index** is an optional data structure that you can create on one or more columns of a table. Indexes can increase the performance of data retrieval.

> **See Also:**
>
> "Introduction to Schema Objects" to learn more about schema object types, storage, and dependencies

## Tables

A table describes an entity such as employees.

You define a table with a table name, such as `employees`, and set of columns. In general, you give each column a name, a data type, and a width when you create the table.

A table is a set of rows. A column identifies an attribute of the entity described by the table, whereas a row identifies an instance of the entity. For example, attributes of the employees

entity correspond to columns for employee ID and last name. A row identifies a specific employee.

You can optionally specify a rule, called an integrity constraint, for a column. One example is a `NOT NULL` integrity constraint. This constraint forces the column to contain a value in every row.

> **See Also:**
>
> - "Overview of Tables" to learn about columns and rows, data types, table storage, and table compression
> - "Data Integrity" to learn about the possible types and states of constraints

## Indexes

An **index** is an optional data structure that you can create on one or more columns of a table. Indexes can increase the performance of data retrieval.

When processing a request, the database can use available indexes to locate the requested rows efficiently. Indexes are useful when applications often query a specific row or range of rows.

Indexes are logically and physically independent of the data. Thus, you can drop and create indexes with no effect on the tables or other indexes. All applications continue to function after you drop an index.

> **See Also:**
>
> "Introduction to Indexes" to learn about the purpose and types of indexes

# Data Access

A general requirement for a DBMS is to adhere to accepted industry standards for a data access language.

- Structured Query Language (SQL)
  SQL is a set-based declarative language that provides an interface to an RDBMS such as Oracle Database.
- PL/SQL and Java
  **PL/SQL** is a procedural extension to Oracle SQL.

## Structured Query Language (SQL)

SQL is a set-based declarative language that provides an interface to an RDBMS such as Oracle Database.

Procedural languages such as C describe *how* things should be done. SQL is nonprocedural and describes *what* should be done.

SQL is the ANSI standard language for relational databases. All operations on the data in an Oracle database are performed using SQL statements. For example, you use SQL to create tables and query and modify data in tables.

A SQL statement can be thought of as a very simple, but powerful, computer program or instruction. Users specify the result that they want (for example, the names of employees), not how to derive it. A SQL statement is a string of SQL text such as the following:

```
SELECT first_name, last_name FROM employees;
```

SQL statements enable you to perform the following tasks:

- Query data
- Insert, update, and delete rows in a table
- Create, replace, alter, and drop objects
- Control access to the database and its objects
- Guarantee database consistency and integrity

SQL unifies the preceding tasks in one consistent language. Oracle SQL is an implementation of the ANSI standard. Oracle SQL supports numerous features that extend beyond standard SQL.

> **See Also:**
>
> "SQL" to learn more about SQL standards and the main types of SQL statements

## PL/SQL and Java

**PL/SQL** is a procedural extension to Oracle SQL.

PL/SQL is integrated with Oracle Database, enabling you to use all of the Oracle Database SQL statements, functions, and data types. You can use PL/SQL to control the flow of a SQL program, use variables, and write error-handling procedures.

A primary benefit of PL/SQL is the ability to store application logic in the database itself. A PL/SQL procedure or function is a schema object that consists of a set of SQL statements and other PL/SQL constructs, grouped together, stored in the database, and run as a unit to solve a specific problem or to perform a set of related tasks. The principal benefit of server-side programming is that built-in functionality can be deployed anywhere.

Oracle Database can also store program units written in Java. A Java stored procedure is a Java method published to SQL and stored in the database for general use. You can call existing PL/SQL programs from Java and Java programs from PL/SQL.

> **See Also:**
>
> - "Server-Side Programming: PL/SQL and Java"
> - *Oracle Database Development Guide* to learn more about choosing a programming environment

# Transaction Management

Oracle Database is designed as a multiuser database. The database must ensure that multiple users can work concurrently without corrupting one another's data.

- Transactions
  A **transaction** is a logical, atomic unit of work that contains one or more SQL statements.
- Data Concurrency
  A requirement of a multiuser RDBMS is the control of **data concurrency**, which is the simultaneous access of the same data by multiple users.
- Data Consistency
  In Oracle Database, each user must see a consistent view of the data, including visible changes made by a user's own transactions and committed transactions of other users.

## Transactions

A **transaction** is a logical, atomic unit of work that contains one or more SQL statements.

An RDBMS must be able to group SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone.

An illustration of the need for transactions is a funds transfer from a savings account to a checking account. The transfer consists of the following separate operations:

1. Decrease the savings account.
2. Increase the checking account.
3. Record the transaction in the transaction journal.

Oracle Database guarantees that all three operations succeed or fail as a unit. For example, if a hardware failure prevents a statement in the transaction from executing, then the other statements must be rolled back.

Transactions are one feature that set Oracle Database apart from a file system. If you perform an atomic operation that updates several files, and if the system fails halfway through, then the files will not be consistent. In contrast, a transaction moves an Oracle database from one consistent state to another. The basic principle of a transaction is "all or nothing": an atomic operation succeeds or fails as a whole.

> **See Also:**
>
> "Transactions " to learn about the definition of a transaction, statement-level atomicity, and transaction control

## Data Concurrency

A requirement of a multiuser RDBMS is the control of **data concurrency**, which is the simultaneous access of the same data by multiple users.

Without concurrency controls, users could change data improperly, compromising data integrity. For example, one user could update a row while a different user simultaneously updates it.

If multiple users access the same data, then one way of managing concurrency is to make users wait. However, the goal of a DBMS is to reduce wait time so it is either nonexistent or negligible. All SQL statements that modify data must proceed with as little interference as possible. Destructive interactions, which are interactions that incorrectly update data or alter underlying data structures, must be avoided.

Oracle Database uses locks to control concurrent access to data. A lock is a mechanism that prevents destructive interaction between transactions accessing a shared resource. Locks help ensure data integrity while allowing maximum concurrent access to data.

> **See Also:**
>
> "Overview of the Oracle Database Locking Mechanism"

## Data Consistency

In Oracle Database, each user must see a consistent view of the data, including visible changes made by a user's own transactions and committed transactions of other users.

For example, the database must prevent the dirty read problem, which occurs when one transaction sees uncommitted changes made by another concurrent transaction.

Oracle Database always enforces statement-level read consistency, which guarantees that the data that a single query returns is committed and consistent for a single point in time. Depending on the transaction isolation level, this point is the time at which the statement was opened or the time the transaction began. The Oracle Flashback Query feature enables you to specify this point in time explicitly.

The database can also provide read consistency to all queries in a transaction, known as transaction-level read consistency. In this case, each statement in a transaction sees data from the same point in time, which is the time at which the transaction began.

> **See Also:**
>
> - "Data Concurrency and Consistency " to learn more about dirty reads
> - *Oracle Database Development Guide* to learn about Oracle Flashback Query

# Oracle Database Architecture

A **database server** is the key to information management.

In general, a server reliably manages a large amount of data in a multiuser environment so that users can concurrently access the same data. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.

- Database and Instance
  An Oracle database server consists of a database and at least one **database instance**, commonly referred to as simply an **instance**.
- Database Storage Structures
  A database can be considered from both a physical and logical perspective.
- Database Instance Structures
  An Oracle database uses memory structures and processes to manage and access the CDB. All memory structures exist in the main memory of the computers that constitute the RDBMS.
- Application and Networking Architecture
  To take full advantage of a given computer system or network, Oracle Database enables processing to be split between the database server and the client programs. The computer running the RDBMS handles the database server responsibilities while the computers running the applications handle the interpretation and display of data.

## Database and Instance

An Oracle database server consists of a database and at least one **database instance**, commonly referred to as simply an **instance**.

Because an instance and a database are so closely connected, the term **Oracle database** sometimes refers to both instance and database. In the strictest sense, the terms have the following meanings:

- Database

  A database is a set of files, located on disk, that store user data. These data files can exist independently of a database instance. Starting in Oracle Database 21c, "database" refers specifically to the data files of a **multitenant container database (CDB)**, **pluggable database (PDB)**, or **application container**.
- Database instance

  An instance is a named set of memory structures that manage database files. A database instance consists of a shared memory area, called the **system global area (SGA)**, and a set of background processes. An instance can exist independently of database files.
- Multitenant Architecture
  The **multitenant architecture** enables an Oracle database to be a CDB.
- Sharding Architecture
  Oracle Sharding is a database scaling technique based on horizontal partitioning of data across multiple PDBs. Applications perceive the pool of PDBs as a single logical database.

## Multitenant Architecture

The **multitenant architecture** enables an Oracle database to be a CDB.

Every Oracle database must contain or be able to be contained by another database. For example, a CDB contains PDBs, and an application container contains application PDBs. A PDB is contained by a CDB or application container, and an application container is contained by a CDB.

Starting in Oracle Database 21c, a multitenant container database is the only supported architecture. In previous releases, Oracle supported non-container databases (non-CDBs).

- CDBs
  A CDB contains one or more user-created PDBs and application containers.
- PDBs
  A **PDB** is a portable collection of schemas, schema objects, and nonschema objects that appears to an application as a separate database.
- Application Containers
  An **application container** is an optional, user-created container within a CDB that stores data and metadata for one or more applications.

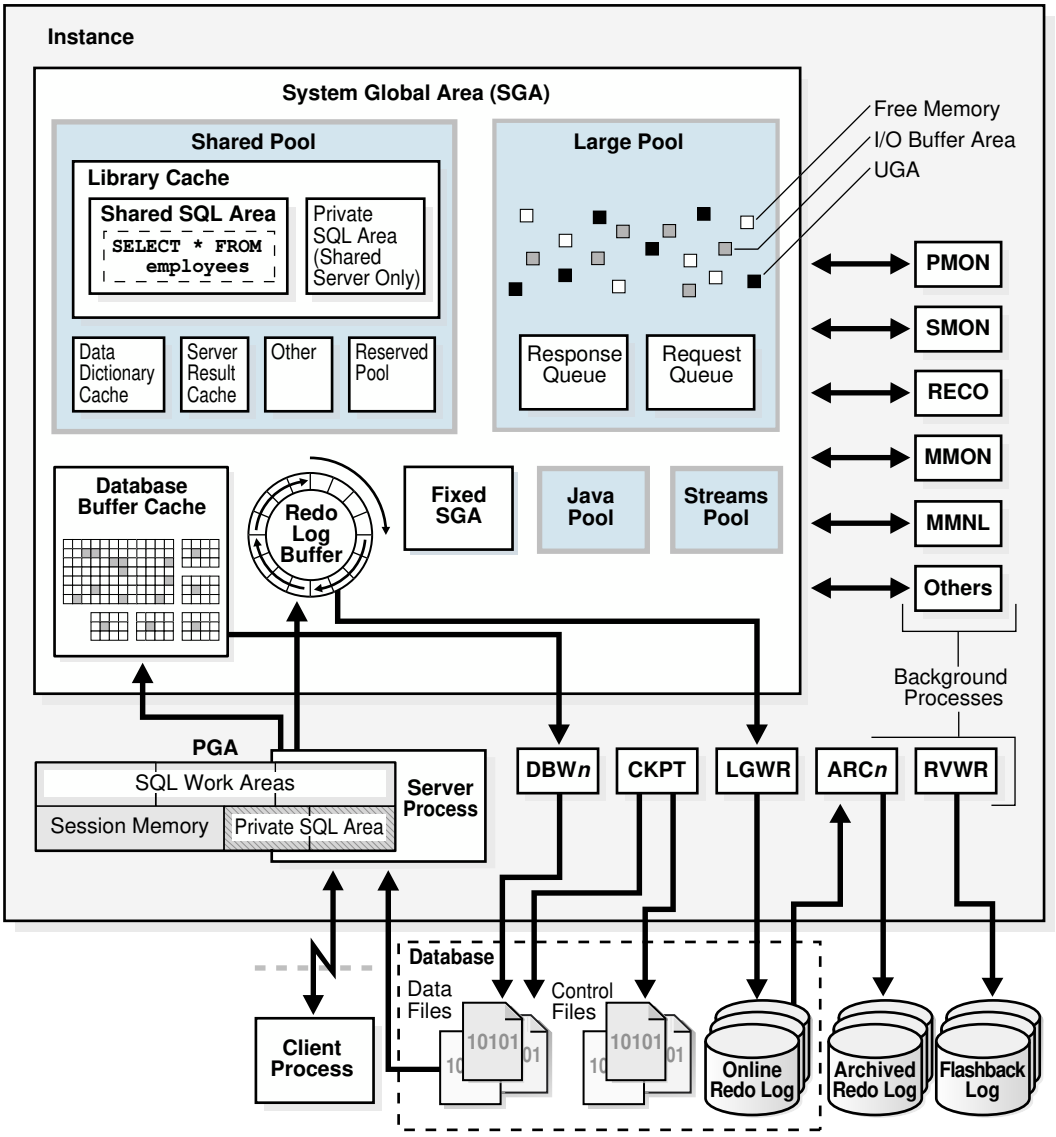
## CDBs

A CDB contains one or more user-created PDBs and application containers.

At the physical level, a CDB is a set of files: control file, online redo log files, and data files. The database instance manages the files that make up the CDB.

The following figure shows a CDB and an associated database instance.

**Figure 1-1 Database Instance and CDB**




## PDBs

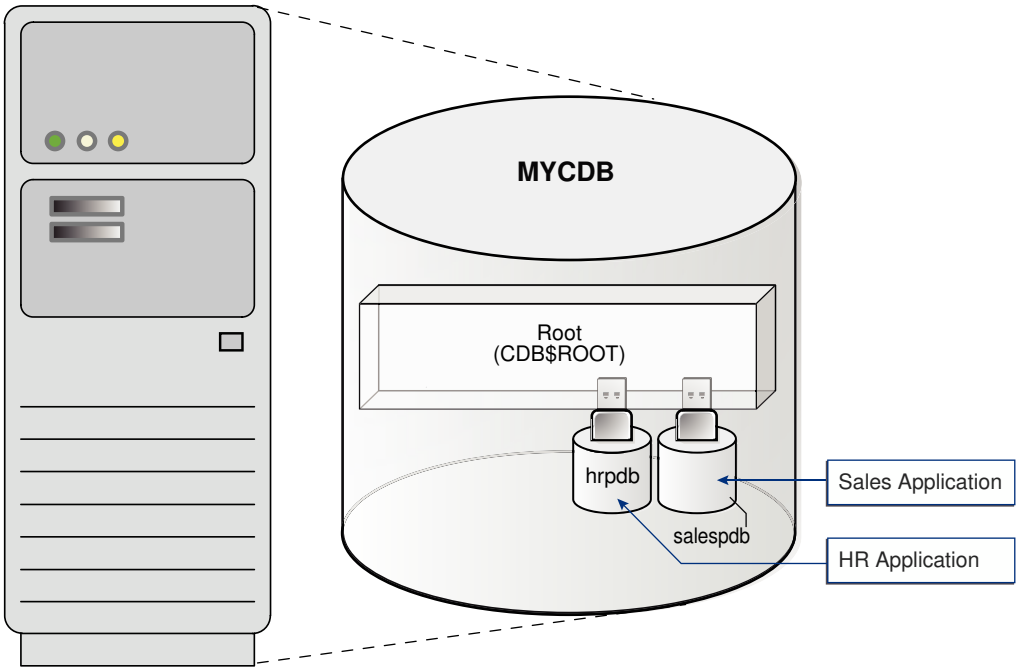
A **PDB** is a portable collection of schemas, schema objects, and nonschema objects that appears to an application as a separate database.

At the physical level, each PDB has its own set of data files that store the data for the PDB. The CDB includes all the data files for the PDBs contained within it, and a set of system data files that store metadata for the CDB itself.

To move or archive a PDB, you can unplug it. An unplugged PDB consists of the PDB data files and a metadata file. An unplugged PDB is not usable until it is plugged in to a CDB.

The following figure shows a CDB named `MYCDB`.

**Figure 1-2 PDBs in a CDB**




Physically, `MYCDB` is an Oracle database, in the sense of a set of data files associated with an instance. `MYCDB` has one database instance, although multiple instances are possible in Oracle Real Application Clusters, and one set of database files.

`MYCDB` contains two PDBs: `hrpdb` and `salespdb`. As shown in Figure 1-2, these PDBs appear to their respective applications as separate, independent databases. An application has no knowledge of whether it is connecting to a CDB or PDB.

To administer the CDB itself or any PDB within it, you can connect to the CDB root. The root is a collection of schemas, schema objects, and nonschema objects to which all PDBs and application containers belong.

## Application Containers

An **application container** is an optional, user-created container within a CDB that stores data and metadata for one or more applications.

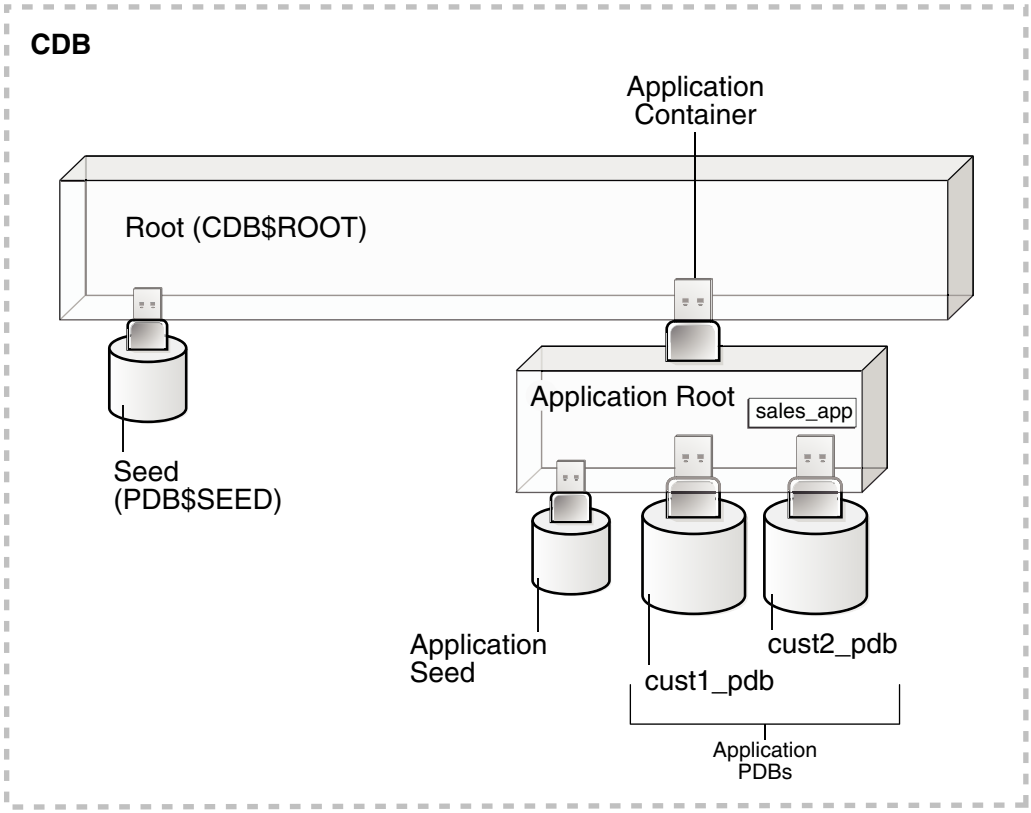
In this context, an application (also called the *master application definition*) is a named, versioned set of common data and metadata stored in the application root. For example, the application might include definitions of tables, views, user accounts, and PL/SQL packages that are common to a set of PDBs.

In some ways, an application container functions as an application-specific CDB *within* a CDB. An application container, like the CDB itself, can include multiple application PDBs, and enables these PDBs to share metadata and data. At the physical level, an application container has its own set of data files, just like a PDB.

For example, a SaaS deployment can use multiple application PDBs, each for a separate customer, which share application metadata and data. For example, in the following figure, `sales_app` is the application model in the application root. The application PDB named `cust1_pdb` contains sales data only for customer 1, whereas the application PDB named

`cust2_pdb` contains sales data only for customer 2. Plugging, unplugging, cloning, and other PDB-level operations are available for individual customer PDBs.

**Figure 1-3 SaaS Use Case**




## Sharding Architecture

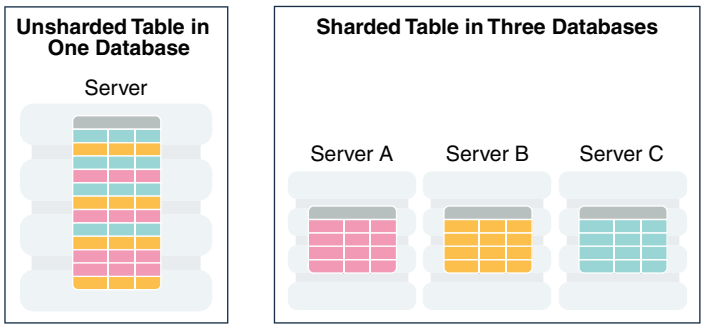
Oracle Sharding is a database scaling technique based on horizontal partitioning of data across multiple PDBs. Applications perceive the pool of PDBs as a single logical database.

Key benefits of sharding for OLTP applications include linear scalability, fault containment, and geographical data distribution. Sharding is well suited to deployment in the Oracle Cloud. Unlike NoSQL data stores that implement sharding, Oracle Sharding provides the benefits of sharding without sacrificing the capabilities of an enterprise RDBMS.

In a sharding architecture, each CDB is hosted on a dedicated server with its own local resources: CPU, memory, flash, or disk. You can designate a PDB as a **shard**. PDB shards from different CDBs make up a single logical database, which is referred to as a **sharded database**. Two shards in the same CDB cannot be members of the *same* sharded database. However, within the same CDB, one PDB could be in one sharded database, and another PDB could be in a separate sharded database.

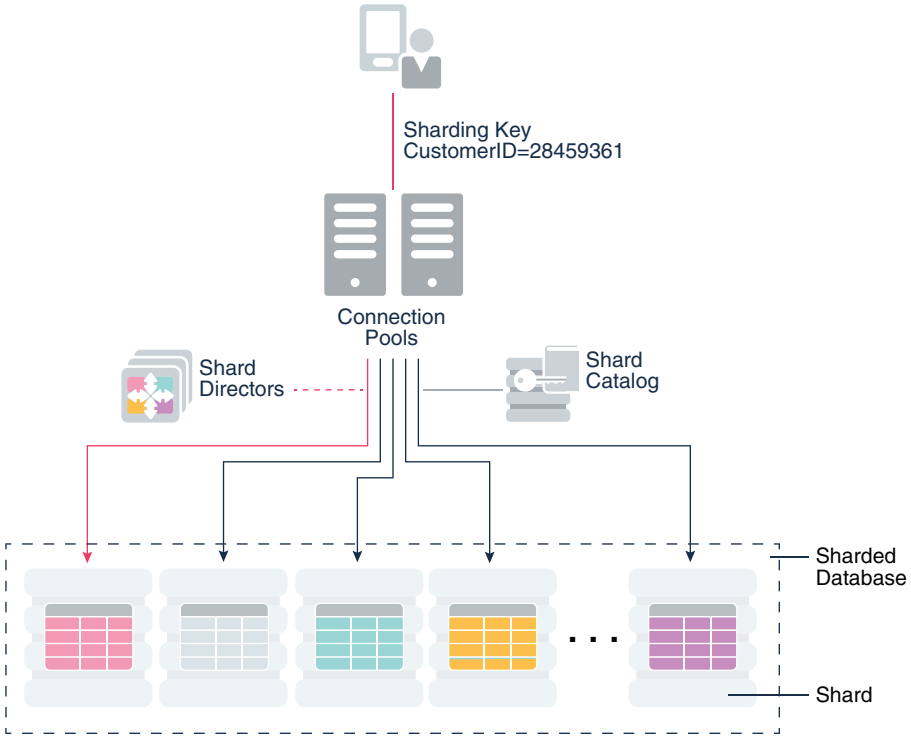
Horizontal partitioning involves splitting a database table across shards so that each shard contains the table with the same columns but a different subset of rows. A table split up in this manner is also known as a **sharded table**. The following figure shows a sharded table horizontally partitioned across three shards, each of which is a PDB in a separate CDB.

**Figure 1-4 Horizontal Partitioning of a Table Across Shards**




A use case is distributing customer account data across multiple CDBs. For example, a customer with ID 28459361 may look up his records. The following figure shows a possible architecture. The customer request is routed through a connection pool, where sharding directors (network listeners) direct the request to the appropriate PDB shard, which contains all the customer rows.

**Figure 1-5 Oracle Sharding Architecture**




**See Also:**

*Using Oracle Sharding* for an overview of Oracle Sharding

# Database Storage Structures

A database can be considered from both a physical and logical perspective.

Physical data is data viewable at the operating system level. For example, operating system utilities such as the Linux `ls` and `ps` can list database files and processes. Logical data such as a table is meaningful only for the database. A SQL statement can list the tables in an Oracle database, but an operating system utility cannot.

The database has physical structures and logical structures. Because the physical and logical structures are separate, you can manage the physical storage of data without affecting access to logical storage structures. For example, renaming a physical database file does not rename the tables whose data is stored in this file.

- Physical Storage Structures
  The physical database structures are the files that store the data.
- Logical Storage Structures
  Logical storage structures enable Oracle Database to have fine-grained control of disk space use.

## Physical Storage Structures

The physical database structures are the files that store the data.

When you execute a `CREATE DATABASE` command, you create a CDB. The following files are created:

- Data files

  Every CDB has one or more physical data files, which contain all the database data. The data of logical database structures, such as tables and indexes, is physically stored in the data files.
- Control files

  Every CDB has a control file. A control file contains metadata specifying the physical structure of the database, including the database name and the names and locations of the database files.
- Online redo log files

  Every CDB has an online redo log, which is a set of two or more online redo log files. An online redo log is made up of redo entries (also called redo log records), which record all changes made to data.

When you execute a `CREATE PLUGGABLE DATABASE` command within a CDB, you create a PDB. The PDB contains a dedicated set of data files within the CDB. A PDB does not have a separate, dedicated control file and online redo log: these files are shared by the PDBs.

Many other files are important for the functioning of a CDB. These include parameter files and networking files. Backup files and archived redo log files are offline files important for backup and recovery.

> **See Also:**
>
> "Physical Storage Structures"

## Logical Storage Structures

Logical storage structures enable Oracle Database to have fine-grained control of disk space use.

This topic discusses logical storage structures:

- Data blocks

  At the finest level of granularity, Oracle Database data is stored in data blocks. One data block corresponds to a specific number of bytes on disk.

- Extents

  An extent is a specific number of logically contiguous data blocks, obtained in a single allocation, used to store a specific type of information.

- Segments

  A segment is a set of extents allocated for a user object (for example, a table or index), undo data, or temporary data.

- Tablespaces

  A database is divided into logical storage units called tablespaces. A tablespace is the logical container for segments. Each tablespace consists of at least one data file.

> **See Also:**
>
> "Logical Storage Structures"

# Database Instance Structures

An Oracle database uses memory structures and processes to manage and access the CDB. All memory structures exist in the main memory of the computers that constitute the RDBMS.

When applications connect to a CDB or PDB, they connect to a database instance. The instance services applications by allocating other memory areas in addition to the SGA, and starting other processes in addition to background processes.

- Oracle Database Processes
  A **process** is a mechanism in an operating system that can run a series of steps. Some operating systems use the terms *job*, *task*, or *thread*.
- Instance Memory Structures
  Oracle Database creates and uses memory structures for program code, data shared among users, and private data areas for each connected user.

## Oracle Database Processes

A **process** is a mechanism in an operating system that can run a series of steps. Some operating systems use the terms *job*, *task*, or *thread*.

For the purposes of this topic, a thread is equivalent to a process. An Oracle database instance has the following types of processes:

- Client processes

These processes are created and maintained to run the software code of an application program or an Oracle tool. Most environments have separate computers for client processes.

- Background processes

  These processes consolidate functions that would otherwise be handled by multiple Oracle Database programs running for each client process. Background processes asynchronously perform I/O and monitor other Oracle Database processes to provide increased parallelism for better performance and reliability.

- Server processes

  These processes communicate with client processes and interact with Oracle Database to fulfill requests.

Oracle processes include server processes and background processes. In most environments, Oracle processes and client processes run on separate computers.

> **See Also:**
>
> "Process Architecture"

## Instance Memory Structures

Oracle Database creates and uses memory structures for program code, data shared among users, and private data areas for each connected user.

The following memory structures are associated with a database instance:

- System Global Area (SGA)

  The SGA is a group of shared memory structures that contain data and control information for one database instance. Examples of SGA components include the database buffer cache and shared SQL areas. The SGA can contain an optional In-Memory Column Store (IM column store), which enables data to be populated in memory in a columnar format.

- Program Global Areas (PGA)

  A PGA is a memory region that contains data and control information for a server or background process. Access to the PGA is exclusive to the process. Each server process and background process has its own PGA.

> **See Also:**
>
> "Memory Architecture"

# Application and Networking Architecture

To take full advantage of a given computer system or network, Oracle Database enables processing to be split between the database server and the client programs. The computer running the RDBMS handles the database server responsibilities while the computers running the applications handle the interpretation and display of data.

- Application Architecture
  The application architecture is the computing environment in which a database application connects to an Oracle database. The two most common database architectures are client/server and multitier.
- Oracle Net Services Architecture
  **Oracle Net Services** is the interface between the database and the network communication protocols that facilitate distributed processing and distributed databases.

## Application Architecture

The application architecture is the computing environment in which a database application connects to an Oracle database. The two most common database architectures are client/server and multitier.

### Client-Server Architecture

In a client/server architecture, the client application initiates a request for an operation to be performed on the database server. The server runs Oracle Database software and handles the functions required for concurrent, shared data access. The server receives and processes requests that originate from clients.

### Multitier Architecture

In a multitier architecture, one or more application servers perform parts of the operation. An application server contains a large part of the application logic, provides access to the data for the client, and performs some query processing. In this way, the load on the database decreases. The application server can serve as an interface between clients and multiple databases and provide an additional level of security.

A service-oriented architecture (SOA) is a multitier architecture in which application functionality is encapsulated in services. SOA services are usually implemented as Web services. Web services are accessible through HTTP and are based on XML-based standards such as Web Services Description Language (WSDL) and SOAP. Oracle Database can act as a Web service provider in a traditional multitier or SOA environment.

**Simple Oracle Document Access (SODA)** is an adaption of SOA that enables you to access to data stored in the database. SODA is designed for schemaless application development without knowledge of relational database features or languages such as SQL and PL/SQL. You can create and store collections of documents in Oracle Database, retrieve them, and query them, without needing to know how the documents are stored. SODA for REST uses the representational state transfer (REST) architectural style to implement SODA.

> **See Also:**
>
> - "Overview of Multitier Architecture"
> - *Oracle XML DB Developer's Guide* for more information about using Web services with the database

## Oracle Net Services Architecture

**Oracle Net Services** is the interface between the database and the network communication protocols that facilitate distributed processing and distributed databases.

Communication protocols define the way that data is transmitted and received on a network. Oracle Net Services supports communications on all major network protocols, including TCP/IP, HTTP, FTP, and WebDAV.

Oracle Net, a component of Oracle Net Services, establishes and maintains a network session from a client application to a database server. After a network session is established, Oracle Net acts as the data courier for both the client application and the database server, exchanging messages between them. Oracle Net can perform these jobs because it is located on each computer in the network.

An important component of Net Services is the Oracle Net Listener (called the *listener*), which is a process that runs on the database or elsewhere in the network. Client applications send connection requests to the listener, which manages the traffic of these requests to the database. When a connection is established, the client and database communicate directly.

The most common ways to configure an Oracle database to service client requests are:

- Dedicated server architecture

  Each client process connects to a dedicated server process. The server process is not shared by any other client for the duration of the client's session. Each new session is assigned a dedicated server process.
- Shared server architecture

  The database uses a pool of shared server processes for multiple sessions. A client process communicates with a dispatcher, which is a process that enables many clients to connect to the same database instance without the need for a dedicated server process for each client.

> **See Also:**
>
> - "Overview of Oracle Net Services Architecture"
> - *Oracle Database Net Services Administrator's Guide* to learn more about Oracle Net architecture
> - *Oracle XML DB Developer's Guide* for information about using WebDAV with the database

# Oracle Database Documentation Roadmap

The documentation set is designed with specific access paths to ensure that users are able to find the information they need as efficiently as possible.

The documentation set is divided into three layers or groups: basic, intermediate, and advanced. Users begin with the manuals in the basic group, proceed to the manuals in the intermediate group (the *2 Day* + series), and finally to the advanced manuals, which include the remainder of the documentation.

You can find the documentation for supported releases of Oracle Database at https://docs.oracle.com/en/database/oracle/oracle-database/.

- Oracle Database Documentation: Basic Group
  Technical users who are new to Oracle Database begin by reading one or more manuals in the basic group from cover to cover. Each manual in this group is designed to be read in two days.

- Oracle Database Documentation: Intermediate Group
  The next step up from the basic group is the intermediate group.
- Oracle Database Documentation: Advanced Group
  The advanced group manuals are intended for expert users who require more detailed information about a particular topic than can be provided by the *2 Day* + manuals.

## Oracle Database Documentation: Basic Group

Technical users who are new to Oracle Database begin by reading one or more manuals in the basic group from cover to cover. Each manual in this group is designed to be read in two days.

In addition to this manual, the basic group includes the manuals shown in the following table.

**Table 1-1 Basic Group**

| Manual | Description |
|---|---|
| *Oracle Database 2 Day Developer's Guide* | This task-based quick start guide explains how to use the basic features of Oracle Database through SQL and PL/SQL. |

The manuals in the basic group are closely related, which is reflected in the number of cross-references. For example, *Oracle Database Concepts* frequently sends users to a *2 Day* manual to learn how to perform a task based on a concept. The *2 Day* manuals frequently reference *Oracle Database Concepts* for conceptual background about a task.

## Oracle Database Documentation: Intermediate Group

The next step up from the basic group is the intermediate group.

Manuals in the intermediate group are prefixed with the word *2 Day* + because they expand on and assume information contained in the *2 Day* manuals. The *2 Day* + manuals cover topics in more depth than is possible in the basic manuals, or cover topics of special interest. The manuals are intended for different audiences:

- Database administrators

  *Oracle Database 2 Day + Performance Tuning Guide* is a quick start guide that describes how to perform day-to-day database performance tuning tasks using features provided by Oracle Diagnostics Pack, Oracle Tuning Pack, and Oracle Enterprise Manager Cloud Control (Cloud Control).
- Database developers

  *Oracle Database 2 Day + Java Developer's Guide* helps you understand all Java products used to build a Java application. The manual explains how to use Oracle JDBC Thin driver, Universal Connection Pool (UCP), and Java in the Database (OJVM) in a sample Web application.

## Oracle Database Documentation: Advanced Group

The advanced group manuals are intended for expert users who require more detailed information about a particular topic than can be provided by the *2 Day* + manuals.

The following table lists essential reference manuals in the advanced group.

**Table 1-2 Essential Reference Manuals**

| Manual | Description |
|---|---|
| *Oracle Database SQL Language Reference* | Provides a complete description of the Structured Query Language (SQL) used to manage information in an Oracle Database. |
| *Oracle Database Reference* | Describes database initialization parameters, data dictionary views, dynamic performance views, wait events, and background processes. |
| *Oracle Database PL/SQL Packages and Types Reference* | Describes the PL/SQL packages provided with the Oracle database server. You can use the supplied packages when creating your applications or for ideas in creating your own stored procedures. |

The advanced guides are too numerous to list in this section. The following table lists guides that the majority of expert Oracle DBAs use.

**Table 1-3 Advanced Group for DBAs**

| Manual | Description |
|---|---|
| *Oracle Database Administrator's Guide* | Explains how to perform tasks such as creating and configuring databases, maintaining and monitoring databases, creating schema objects, scheduling jobs, and diagnosing problems. |
| *Oracle Database Security Guide* | Describes how to configure security for Oracle Database by using the default database features. |
| *Oracle Database Performance Tuning Guide* | Describes how to use Oracle Database tools to optimize database performance. This guide also describes performance best practices for creating a database and includes performance-related reference information. |
| *Oracle Database SQL Tuning Guide* | Describes SQL processing, the optimizer, execution plans, SQL operators, optimizer statistics, application tracing, and SQL advisors. |
| *Oracle Database Backup and Recovery User's Guide* | Explains how to back up, restore, and recover Oracle databases, perform maintenance on backups of database files, and transfer data between storage systems. |
| *Oracle Real Application Clusters Administration and Deployment Guide* | Explains how to install, configure, manage, and troubleshoot an Oracle RAC database. |

The following table lists guides that the majority of expert Oracle developers use.

**Table 1-4 Advanced Group for Developers**

| Manual | Description |
|---|---|
| *Oracle Database Development Guide* | Explains how to develop applications or convert existing applications to run in the Oracle Database environment. The manual explains fundamentals of application design, and describes essential concepts for developing in SQL and PL/SQL. |

**Table 1-4 (Cont.) Advanced Group for Developers**

| Manual | Description |
|---|---|
| *Oracle Database PL/SQL Language Reference* | Describes all aspects of the PL/SQL language, including data types, control statements, collections, triggers, packages, and error handling. |
| *Oracle Database Java Developer's Guide* | Describes how to develop, load, and run Java applications in Oracle Database. |
| *Oracle Database SecureFiles and Large Objects Developer's Guide* | Explains how to develop new applications using Large Objects (LOBs), SecureFiles LOBs, and Database File System (DBFS). |

Other advanced guides required by a particular user depend on the area of responsibility of this user.

# Part I
# Multitenant Architecture

The **multitenant architecture** enables an Oracle database to function as a multitenant container database (CDB).

This part contains the following chapters:


- CDBs and PDBs
- Application Containers


- CDBs and PDBs
  The **multitenant architecture** enables an Oracle database to function as a multitenant container database (CDB).
- Application Containers
  Within a CDB, you can create a container for application data and metadata that can be shared by PDBs.

2

# CDBs and PDBs

The **multitenant architecture** enables an Oracle database to function as a multitenant container database (CDB).

Starting in Oracle Database 21c, a multitenant container database is the only supported architecture. In previous releases, Oracle supported non-container databases (non-CDBs).

This chapter contains the following topics:


- About Containers in a CDB
- The CDB Root and System Container
- PDBs


- About Containers in a CDB
  A **container** is a collection of schemas, objects, and related structures in a **multitenant container database (CDB)**. Within a CDB, each container has a unique ID and name.
- The CDB Root and System Container
  The **CDB root**, also called simply *the root*, is a collection of schemas, schema objects, and nonschema objects to which all PDBs belong.
- PDBs
  A **PDB** is a user-created set of schemas, objects, and related structures that appears logically to a client application as a separate database.

## About Containers in a CDB

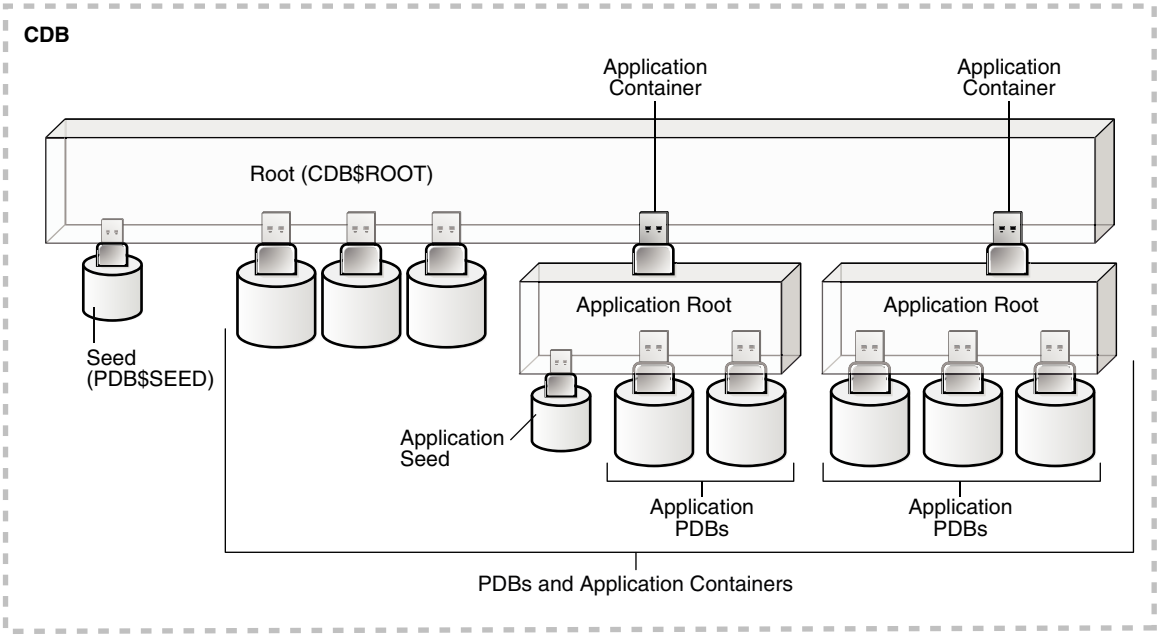
A **container** is a collection of schemas, objects, and related structures in a **multitenant container database (CDB)**. Within a CDB, each container has a unique ID and name.

A **CDB** includes zero, one, or many customer-created pluggable databases (PDBs) and application containers. A **PDB** is a portable collection of schemas, schema objects, and nonschema objects that appears to an Oracle Net client as a separate database. An **application container** is an optional, user-created CDB component that stores data and metadata for one or more application back ends. A CDB includes zero or more application containers.

> **Note:**
>
> "Application Containers" explains application containers in detail.

The following figure represents possible containers in a CDB.

**Figure 2-1 Containers in a CDB**




Every CDB has the following containers:

- Exactly one CDB root container (also called simply *the root*)

  The CDB root is a collection of schemas, schema objects, and nonschema objects to which all PDBs belong (see "CDBs and PDBs"). The root stores Oracle-supplied metadata and common users. An example of metadata is the source code for Oracle-supplied PL/SQL packages. A common user is a database user known in every container (see "Common User Accounts"). The root container is named `CDB$ROOT`.

- Exactly one system container

  The system container includes the root CDB and all PDBs in the CDB. Thus, the system container is the logical container for the CDB itself.

- Zero or more application containers

  An application container consists of exactly one application root, and the PDBs plugged in to this root. Whereas the system container contains the CDB root and *all* the PDBs within the CDB, an application container includes only the PDBs plugged into the application root. An application root belongs to the CDB root and no other container.

- Zero or more user-created PDBs

  A PDB contains the data and code required for a specific set of features (see "PDBs"). For example, a PDB can support a specific application, such as a human resources or sales application. No PDBs exist at creation of the CDB. You add PDBs based on your business requirements.

  A PDB belongs to exactly zero or one application container. If a PDB belongs to an application container, then it is an application PDB. For example, the `cust1_pdb` and `cust2_pdb` application PDBs might belong to the `saas_sales_ac` application container, in which case they belong to no other application containers. An application seed is an optional application PDB that acts as a user-created PDB template, enabling you to create new application PDBs rapidly.

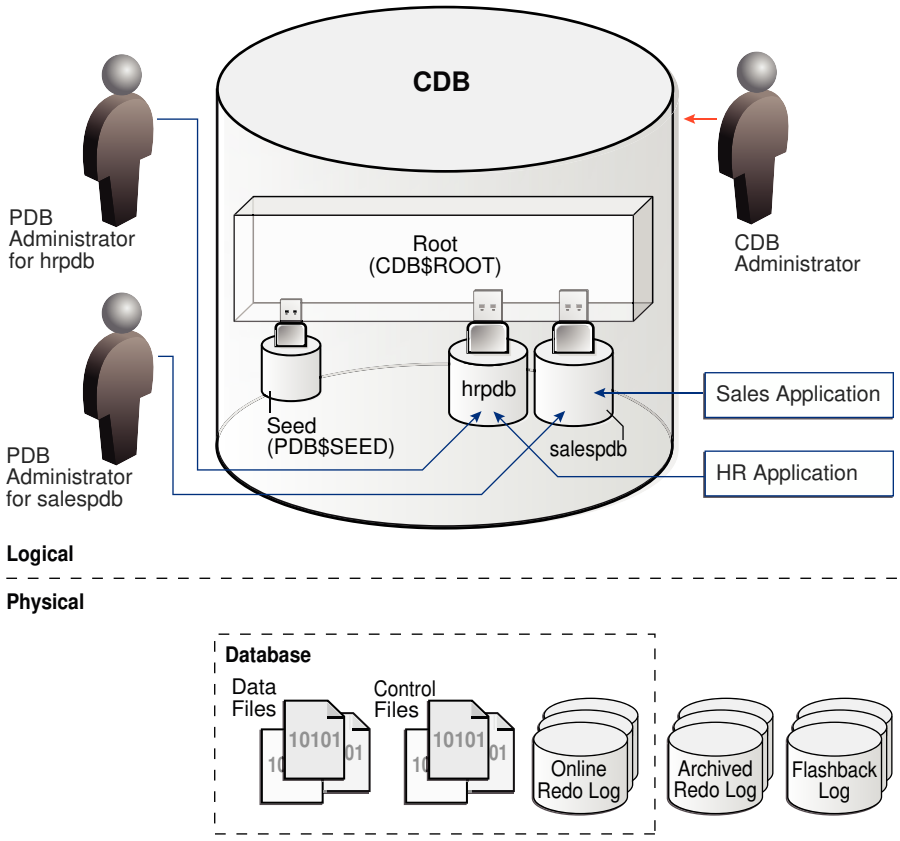
- Exactly one seed PDB

  The seed PDB is a system-supplied template that the CDB can use to create new PDBs. The seed PDB is named `PDB$SEED`. You cannot add or modify objects in `PDB$SEED`.

**Example 2-1 CDB with No Application Containers**

This example shows a simple CDB with five containers: the system container (the entire CDB), the CDB root, the PDB seed (`PDB$SEED`), and two PDBs. Each PDB has its own dedicated application. A different PDB administrator manages each PDB. A common user exists across a CDB with a single identity. In this example, common user `SYS` can manage the root and every PDB. At the physical level, this CDB is managed by one or more database instances, and contains a set of data files for each PDB and for the CDB itself.

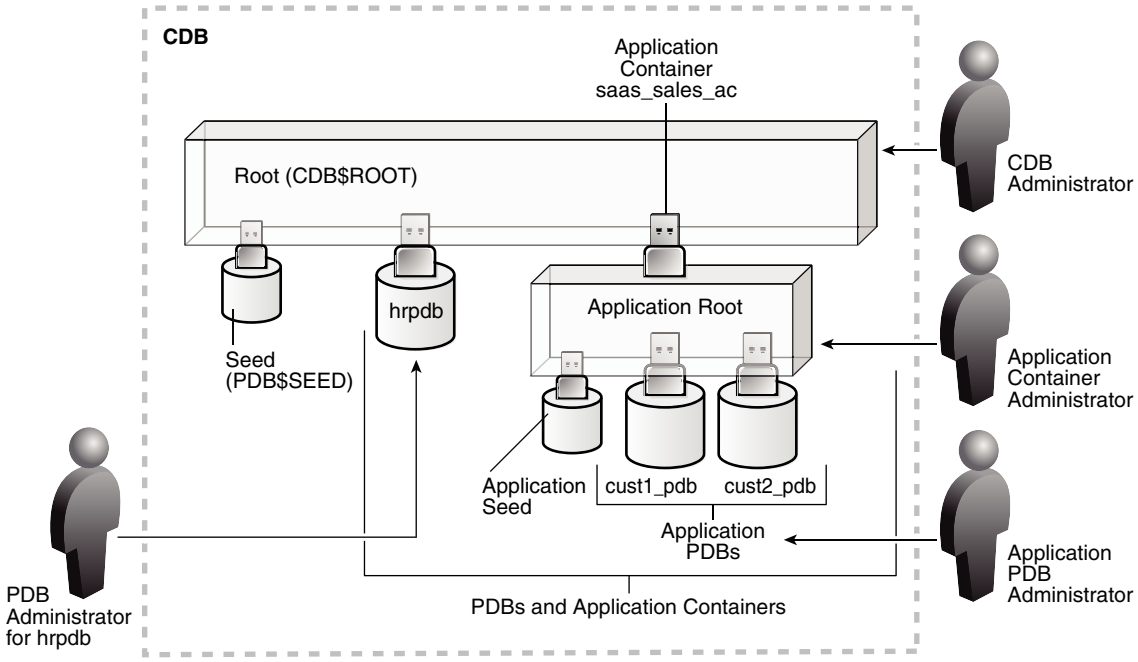
**Figure 2-2 CDB with No Application Containers**




**Example 2-2 CDB with an Application Container**

In this variation, the CDB contains an application container named `saas_sales_ac`. Within the application container, the application PDB `cust1_pdb` supports an application for one customer, and the application PDB `cust2_pdb` supports an application for a different customer. The CDB also contains a PDB named `hrpdb`, which supports an HR application, but does not belong to an application container.

**Figure 2-3 CDB with an Application Container**




In this example, multiple DBAs manage the CDB environment:

- A CDB administrator manages the CDB itself.
- An application container administrator manages the `saas_sales_ac` container, including application installation and upgrades.
- An application PDB administrator manages the two PDBs in the `saas_sales_ac` container: `cust1_pdb` and `cust2_pdb`.
- A PDB administrator manages `hrpdb`.

# The CDB Root and System Container

The **CDB root**, also called simply *the root*, is a collection of schemas, schema objects, and nonschema objects to which all PDBs belong.

Every CDB has one and only one root container named `CDB$ROOT`. The root stores the system metadata required to manage PDBs. All PDBs belong to the root. The system container is the CDB root and all PDBs that belong to this root.

The CDB root does not store user data. Oracle recommends that you do *not* add common objects to the root or modify Oracle-supplied schemas in the root. However, you can create common users and roles for database administration. A common user with the necessary privileges can switch between containers.

Oracle recommends AL32UTF8 for the root character set. PDBs with different character sets can reside in the same CDB without requiring character set conversion.

**Example 2-3 All Containers in a CDB**

The following query, issued by an administrative user connected to the CDB root, lists all containers in the CDB (including the seed and CDB root), ordered by `CON_ID`.

```
COL NAME FORMAT A15
SELECT NAME, CON_ID, DBID, CON_UID, GUID
FROM   V$CONTAINERS ORDER BY CON_ID;

NAME          CON_ID       DBID    CON_UID GUID
------------- ------ ---------- ---------- --------------------------------
CDB$ROOT           1 1895287725          1 2003321EDD4F60D6E0534E40E40A41C5
PDB$SEED           2 2795386505 2795386505 200AC90679F07B55E05396C0E40A23FE
SAAS_SALES_AC      3 1239646423 1239646423 200B4CE0A8DC1D24E05396C0E40AF8EE
SALESPDB           4 3692549634 3692549634 200B4928319C1BCCE05396C0E40A2432
HRPDB              5 3784483090 3784483090 200B4928319D1BCCE05396C0E40A2432
```

> **See Also:**
>
> "Common User Accounts"

# PDBs

A **PDB** is a user-created set of schemas, objects, and related structures that appears logically to a client application as a separate database.

Every PDB is owned by `SYS`, regardless of which user created the PDB. `SYS` is a common user in the CDB, which means that this user that has the same identity in the root and in every existing and future PDB within the CDB.

- Types of PDBs
  All PDBs are user-created with the `CREATE PLUGGABLE DATABASE` statement except for `PDB$SEED`, which is Oracle-supplied.
- Purpose of PDBs
  For an application, a PDB is a self-contained, fully functional Oracle database. You can consolidate PDBs into a single CDB to achieve economies of scale, while maintaining isolation between PDBs.
- Proxy PDBs
  A **proxy PDB** refers to a remote PDB, called the **referenced PDB**.
- Names for PDBs
  Containers in a CDB share the same namespace, which means that they must have unique names within this namespace.
- Database Links Between PDBs
  By default, a user connected to one PDB must use database links to access objects in a different PDB.

# Types of PDBs

All PDBs are user-created with the `CREATE PLUGGABLE DATABASE` statement except for `PDB$SEED`, which is Oracle-supplied.

You can create the following types of PDBs.

### Standard PDB

This type of PDB results from running `CREATE PLUGGABLE DATABASE` *without* specifying the PDB as a seed, proxy PDB, or application root. Its capabilities depend on the container in which you create it:

- PDB plugged in to the CDB root

  This PDB belongs to the CDB root container and not an application container. This type of PDB cannot use application common objects. See "Application Common Objects".

- Application PDB

  An application PDB belongs to exactly one application container. Unlike PDBs plugged in to the CDB root, application PDBs can share a master application definition within an application container. For example, a `usa_zipcodes` table in an application root might be a data-linked common object, which means it contains data accessible by all application PDBs plugged in to this root. PDBs that do not reside within the application container cannot access its application common objects.

### Application Root

Consider an application root as an application-specific root container. It serves as a repository for a master definition of an application back end, including common data and metadata. To create an application root, connect to the CDB root and specify the `AS APPLICATION CONTAINER` clause in a `CREATE PLUGGABLE DATABASE` statement. See "Application Root".

### Seed PDBs

Unlike a standard PDB, a seed PDB is not intended to support an application. Rather, the seed is a *template* for the creation of PDBs that support applications. A seed can be either of the following:

- Seed PDB plugged in the CDB root (`PDB$SEED`)

  You can use this system-supplied template to create new PDBs either in an application container or the system container. The system container contains exactly one PDB seed. You cannot drop `PDB$SEED`, or add objects to or modify objects within it.

- Application seed PDB

  To accelerate creation of application PDBs within an application container, you can create an optional application seed. An application container contains either zero or one application seed.

  You create an application seed by connecting to the application container and executing the `CREATE PLUGGABLE DATABASE ... AS SEED` statement. See "Application Seed".

### Proxy PDBs

A proxy PDB is a PDB that uses a database link to reference a PDB in a remote CDB. When you issue a statement in a proxy PDB while the PDB is open, the statement executes in the referenced PDB.

You must create a proxy PDB while connected to the CDB root or application root. You can alter or drop a proxy PDB just as you can a standard PDB.

## Purpose of PDBs

For an application, a PDB is a self-contained, fully functional Oracle database. You can consolidate PDBs into a single CDB to achieve economies of scale, while maintaining isolation between PDBs.

You can use PDBs to achieve the following specific goals:

- Store data specific to an application

  For example, a sales application can have its own dedicated PDB, and a human resources application can have its own dedicated PDB. Alternatively, you can create an application container, which is a named collection of PDBs, to store an application back end containing common data and metadata.

- Move data into a different CDB

  A database is "pluggable" because you can package it as a self-contained unit, called an unplugged PDB, and then move it into another CDB.

- Perform rapid upgrades

  You can unplug a PDB from CDB at a lower Oracle Database release, and then plug it in to a CDB at a higher release.

- Copy data quickly without loss of availability

  For testing and development, you can clone a PDB while it remains open, storing the clone in the same or a different CDB. Optionally, you can specify the PDB as a refreshable clone PDB. Alternatively, you use the Oracle-supplied seed PDB or a user-created application seed to copy new PDBs.

- Reference data in a different CDB

  You can create a proxy PDB that refers to a different PDB, either in the same CDB or in a separate CDB. When you issue statements in the proxy PDB, they execute in the referenced PDB.

- Isolate grants within PDBs

  A local or common user with appropriate privileges can grant `EXECUTE` privileges on a schema object to `PUBLIC` within an individual PDB.

> **See Also:**
>
> - "About Application Containers"
> - *Oracle Database Security Guide* to learn how to grant roles and privileges in a CDB

## Proxy PDBs

A **proxy PDB** refers to a remote PDB, called the **referenced PDB**.

Although you issue SQL statements in the proxy (referring) PDB, the statements execute in the referenced PDB. In this respect, a proxy PDB is loosely analogous to a symbolic link file in Linux.

Proxy PDBs provide the following benefits:

- Aggregate data from multiple application models

  Proxy PDBs enable you to build location-transparent applications that can aggregate data from multiple sources. These sources can be in the same data center or distributed across data centers.

- Enable an application root in one CDB to propagate application changes to a different application root

  Assume that CDBs `cdb_prod` and `cdb_test` have the same application model. You create a proxy PDB in an application container in `cdb_prod` that refers to an application root in `cdb_test`. When you run installation and upgrade scripts in the application root in `cdb_prod`, Oracle Database propagates these statements to the proxy PDB, which in turn sends them remotely to the application root in `cdb_test`. In this way, the application root in `cdb_test` becomes a replica of the application root in `cdb_prod`.

To create a proxy PDB, execute `CREATE PLUGGABLE DATABASE` with the `AS PROXY FROM` clause, where `FROM` specifies the referenced PDB name and a database link. The creation statement copies only the data files belonging to the `SYSTEM` and `SYSAUX` tablespaces.

**Example 2-4 Creating a Proxy PDB**

This example connects to the container `saas_sales_ac` in a local production CDB. The `sales_admin` common user creates a proxy PDB named `sales_sync_pdb`. This application PDB references an application root named `saas_sales_test_ac` in a remote development CDB, which it accesses using the `cdb_dev_rem` database link. When an application upgrade occurs in `saas_sales_ac` in the production CDB, the upgrade automatically propagates to the application root `saas_sales_test_ac` in the remote development CDB.

```
CONNECT sales_admin@saas_sales_ac
Password: ***********

CREATE PLUGGABLE DATABASE sales_sync_pdb AS PROXY FROM
saas_sales_test_ac@cdb_dev_rem;
```

# Names for PDBs

Containers in a CDB share the same namespace, which means that they must have unique names within this namespace.

Names for the following containers must not conflict within the same CDB:

- The CDB root
- PDBs plugged in to the CDB root
- Application roots
- Application PDBs

For example, if the same CDB contains the application containers `saas_sales_ac` and `saas_sales_test_ac`, then two application PDBs that are both named `cust1` cannot simultaneously reside in both containers. The namespace rules also prevent creation of a PDB named `cust1pdb` in the CDB root and a PDB named `cust1pdb` in an application root.

Names for PDBs and application root containers must follow the same rules as net service names. Moreover, because a PDB or application root has a service with its own name, the container name must be unique across all CDBs whose services are exposed through a specific listener. The first character of a user-created container name must be alphanumeric, with remaining characters either alphanumeric or an underscore (_). Because service names are case-insensitive, container names are case-insensitive, and are in upper case even if specified using delimited identifiers.

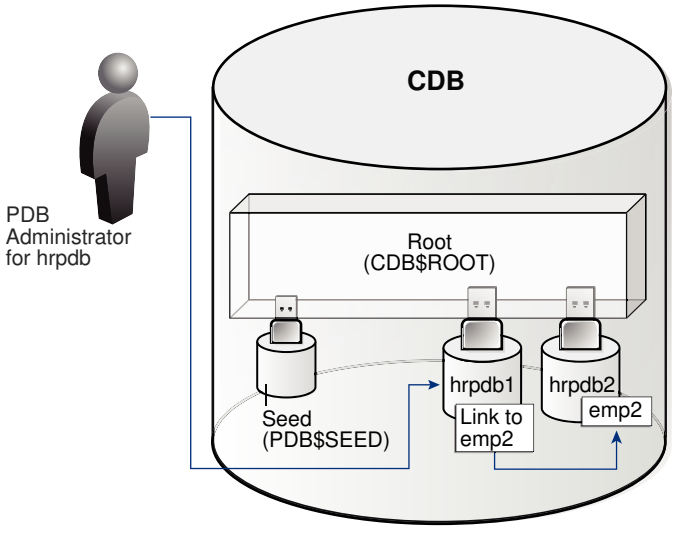
> **See Also:**
>
> *Oracle Database Net Services Reference* for the rules for service names

## Database Links Between PDBs

By default, a user connected to one PDB must use database links to access objects in a different PDB.

**Figure 2-4 Database Link Between PDBs**

In this illustration, a PDB administrator is connected to the PDB named `hrpdb1`. By default, during this user session, `c##dba` cannot query the `emp2` table in `hrpdb2` without specifying a database link.




Exceptions to the rule include:

- A data-linked common object, which is accessible by all application PDBs that contain a data link that points to this object. For example, the application container `saas_sales_ac` might contain the data-linked table `usa_zipcodes` within its application. In this case, common CDB user `c##dba` can connect to an application PDB in this container, and then query `usa_zipcodes` even though the actual table resides in the application root. In this case, no database link is required.

- The `CONTAINERS()` clause in SQL issued from the CDB root or application root. Using this clause, you can query data across all PDBs plugged in to the container root.

When creating a proxy PDB, you must specify a database link name in the `FROM` clause of the `CREATE PLUGGABLE DATABASE ... AS PROXY` statement. If the proxy PDB and referenced PDB reside in separate CDBs, then the database link must be defined in the root of the CDB that will contain the proxy PDB. The database link must connect either to the remote referenced PDB or to the CDB root of the remote CDB.

> **See Also:**
>
> "Common and Local Objects"

3

# Application Containers

Within a CDB, you can create a container for application data and metadata that can be shared by PDBs.

This chapter contains the following topics:


- About Application Containers
- Application Common Objects
- Container Maps
- Cross-Container Operations


- About Application Containers
  An **application container** is an optional, user-created CDB component that stores data and metadata for one or more application back ends. A CDB includes zero or more application containers.
- Application Common Objects
  An **application common object** is a common object created within an application in an application root. Common objects are either data-linked or metadata-linked.
- Container Maps
  A **container map** enables a session connected to application root to issue SQL statements that are routed to the appropriate PDB, depending on the value of a predicate used in the SQL statement.
- Cross-Container Operations
  A **cross-container operation** is a DDL or DML statement that affects multiple containers at once.

> **See Also:**
>
> "Common and Local Objects" to learn about application common objects

## About Application Containers

An **application container** is an optional, user-created CDB component that stores data and metadata for one or more application back ends. A CDB includes zero or more application containers.

Within an **application container**, an **application** is the named, versioned set of common data and metadata stored in the application root. In this context of an application container, the term “application” means “master application definition.” For example, the application might include definitions of tables, views, and packages.

For example, you might create multiple sales-related PDBs within one application container, with these PDBs sharing an application that consists of a set of common tables and table

definitions. You might store multiple HR-related PDBs within a separate application container, with their own common tables and table definitions.

The `CREATE PLUGGABLE DATABASE` statement with the `AS APPLICATION CONTAINER` clause creates the application root of the application container, and thus implicitly creates the application container itself. When you first create the application container, it contains no PDBs. To create application PDBs, you must connect to the application root, and then execute the `CREATE PLUGGABLE DATABASE` statement.

In the `CREATE PLUGGABLE DATABASE` statement, you must specify a container name (which is the same as the application root name), for example, `saas_sales_ac`. The application container name must be unique within the CDB, and within the scope of all the CDBs whose instances are reached through a specific listener. Every application container has a default service with the same name as the application container.

- Purpose of Application Containers
  In some ways, an application container functions as an application-specific CDB *within* a CDB. An application container, like the CDB itself, can include multiple PDBs, and enables these PDBs to share metadata and data.
- Application Root
  An application container has exactly one **application root**, which is the parent of the application PDBs in the container.
- Application PDBs
  An **application PDB** is a PDB that resides in an application container. Every PDB in a CDB resides in either zero or one application containers.
- Application Seed
  An **application seed** is an optional, user-created PDB within an application container. An application container has either zero or one application seed.

# Purpose of Application Containers

In some ways, an application container functions as an application-specific CDB *within* a CDB. An application container, like the CDB itself, can include multiple PDBs, and enables these PDBs to share metadata and data.

The application root enables application PDBs to share an **application**, which in this context means a named, versioned set of common metadata and data. A typical application installs application common users, metadata-linked common objects, and data-linked common objects.

- Key Benefits of Application Containers
  Application containers provide several benefits over storing each application in a separate PDB.
- Application Container Use Case: SaaS
  A SaaS deployment can use multiple application PDBs, each for a separate customer, that share metadata and data.
- Application Containers Use Case: Logical Data Warehouse
  A customer can use multiple application PDBs to address data sovereignty issues.

## Key Benefits of Application Containers

Application containers provide several benefits over storing each application in a separate PDB.

- The application root stores metadata and data that all application PDBs can share.

For example, all application PDBs can share data in a central table, such as a table listed default application roles. Also, all PDBs can share a table definition to which they add PDB-specific rows.

- You maintain your master application definition in the application root, instead of maintaining a separate copy in each PDB.

  If you upgrade the application in the application root, then the changes are automatically propagated to all application PDBs. The application back end might contain the **data-linked common object** `app_roles`, which is a table that list default roles: `admin`, `manager`, `sales_rep`, and so on. A user connected to any application PDB can query this table.

- An application container can include an application seed, application PDBs, and proxy PDBs (which refer to PDBs in other CDBs).
- You can rapidly create new application PDBs from the **application seed**.
- You can query views that report on all PDBs in the application container.
- While connected to the application root, you can use the `CONTAINERS` function to perform DML on objects in multiple PDBs.

  For example, if the `products` table exists in every application PDB, then you can connect to the application root and query the products in all application PDBs using a single `SELECT` statement.

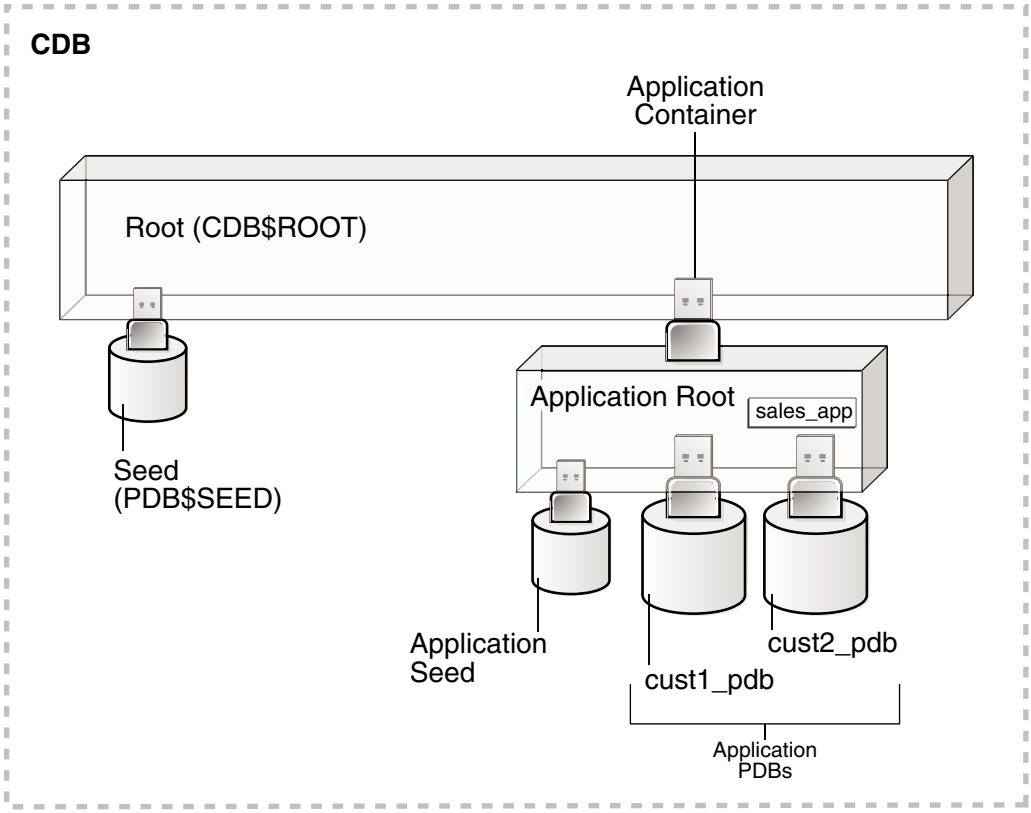
- You can unplug a PDB from an application root, and then plug it in to an application root in a higher Oracle database release. Thus, PDBs are useful in an Oracle database upgrade.

## Application Container Use Case: SaaS

A SaaS deployment can use multiple application PDBs, each for a separate customer, that share metadata and data.

In a pure SaaS environment, the master application definition resides in the application root, but the customer-specific data resides in its own application PDB. For example, `sales_app` is the application model in the application root. The application PDB named `cust1_pdb` contains sales data only for customer 1, whereas the application PDB named `cust2_pdb` contains sales data only for customer 2. Plugging, unplugging, cloning, and other PDB-level operations are available for individual customer PDBs.

**Figure 3-1 SaaS Use Case**




A pure SaaS configuration provides the following benefits:

- Performance
- Security
- Support for multiple customers

  The data for each customer resides in its own container, but is consolidated so that you can manage many customers collectively. This model extends the economies of scale of managing many as one to the application administrator, not only the DBA.

## Application Containers Use Case: Logical Data Warehouse

A customer can use multiple application PDBs to address data sovereignty issues.

In a sample use case, a company puts data specific to each financial quarter in a separate PDB. For example, the application container named `sales_ac` includes `q1_2016_pdb`, `q2_2016_pdb`, `q3_2016_pdb`, and `q4_2016_pdb`. You define each transaction in the PDB corresponding to the associated quarter. To generate a report that aggregates performance across a year, you aggregate across the four PDBs using the `CONTAINERS()` clause.

Benefits of this logical warehouse design include:

- ETL for data specific to a single PDB does not affect the other PDBs.
- Execution plans are more efficient because they are based on actual data distribution.

# Application Root

An application container has exactly one **application root**, which is the parent of the application PDBs in the container.

The property of being an application root is established at creation time, and cannot be changed. The only container to which an application root belongs is the CDB root. An application root is like the CDB root in some ways, and like a PDB in other ways:

- Like the CDB root, an application root serves as parent container to the PDBs plugged into it. When connected to the application root, you can manage common users and privileges, create application PDBs, switch containers, and issue DDL that applies to all PDBs in the application container.
- Like a PDB, you create an application root with the `CREATE PLUGGABLE DATABASE` statement, alter it with `ALTER PLUGGABLE DATABASE`, and change its availability with `STARTUP` and `SHUTDOWN`. You can use DDL to plug, unplug, and drop application roots. The application root has its own service name, and users can connect to the application root in the same way that they connect to a PDB.

An application root differs from both the CDB root and standard PDB because it can store *user-created* common objects, which are called **application common objects**. Application common objects are accessible to the application PDBs plugged in to the application root. Application common objects are not visible to the CDB root, other application roots, or PDBs that do not belong to the application root.

**Example 3-1 Creating an Application Root**

In this example, you log in to the CDB root as administrative common user `c##system`. You create an application container named `saas_sales_ac`, and then open the application root, which has the same name as the container.

```
-- Create the application container called saas_sales_ac
CREATE PLUGGABLE DATABASE saas_sales_ac AS APPLICATION CONTAINER
  ADMIN USER saas_sales_ac_adm IDENTIFIED BY manager;

-- Open the application root
ALTER PLUGGABLE DATABASE saas_sales_ac OPEN;
```

You set the current container to `saas_sales_ac`, and then verify that this container is the application root:

```
-- Set the current container to saas_sales_ac
ALTER SESSION SET CONTAINER = saas_sales_ac;

COL NAME FORMAT a15
COL ROOT FORMAT a4
SELECT CON_ID, NAME, APPLICATION_ROOT AS ROOT,
       APPLICATION_PDB AS PDB,
FROM   V$CONTAINERS;

    CON_ID NAME            ROOT PDB
---------- --------------- ---- ---
         3 SAAS_SALES_AC   YES    NO
```

## Application PDBs

An **application PDB** is a PDB that resides in an application container. Every PDB in a CDB resides in either zero or one application containers.

For example, the `saas_sales_ac` application container might support multiple customers, with each customer application storing its data in a separate PDB. The application PDBs `cust1_sales_pdb` and `cust2_sales_pdb` might reside in `saas_sales_ac`, in which case they belong to no other application container (although as PDBs they necessarily belong also to the CDB root).

Create an application PDB by executing `CREATE PLUGGABLE DATABASE` while connected to the application root. You can either create the application PDB from a seed, or clone a PDB or plug in an unplugged PDB. Like a PDB that is plugged in to CDB root, you can clone, unplug, or drop an application PDB. However, an application PDB must always belong to an application root.

## Application Seed

An **application seed** is an optional, user-created PDB within an application container. An application container has either zero or one application seed.

An application seed enables you to create application PDBs quickly. It serves the same role within the application container as `PDB$SEED` serves within the CDB itself.

The application seed name is always *`application_container_name`*`$SEED`, where *`application_container_name`* is the name of the application container. For example, use the `CREATE PDB ... AS SEED` statement to create `saas_sales_ac$SEED` in the `saas_sales_ac` application container.

# Application Common Objects

An **application common object** is a common object created within an application in an application root. Common objects are either data-linked or metadata-linked.

For a data-linked common object, application PDBs share a single set of data. For example, an application for the `saas_sales_ac` application container is named `saas_sales_app`, has version `1.0`, and includes a data-linked `usa_zipcodes` table. In this case, the rows are stored once in the table in the application root, but are visible in all application PDBs.

For a metadata-linked common object, application PDBs share only the metadata, but contain different sets of data. For example, a metadata-linked `products` table has the same definition in every application PDB, but the rows themselves are specific to the PDB. The application PDB named `cust1pdb` might have a `products` table that contains books, whereas the application PDB named `cust2pdb` might have a `products` table that contains auto parts.

- About Commonality in a CDB
  A common phenomenon defined in a CDB or application root is the same in all containers plugged in to this root.
- Creation of Application Common Objects
  To create common objects, connect to an application root, and then execute a `CREATE` statement that specifies a sharing attribute.

- Metadata-Linked Application Common Objects
  A **metadata link** is a dictionary object that supports referring to, and granting privileges on, common metadata shared by all PDBs in the application container.
- Data-Linked Application Common Objects
  A **data-linked object** is an object whose metadata and data reside in an application root, and are accessible from all application PDBs in this application container.
- Extended Data-Linked Application Objects
  An **extended data-linked object** is a hybrid of a data-linked object and metadata-linked object.

> **See Also:**
>
> "Common and Local Objects" to learn about common objects

# About Commonality in a CDB

A common phenomenon defined in a CDB or application root is the same in all containers plugged in to this root.

- Principles of Commonality
  In a CDB, a phenomenon can be common within either the system container (the CDB itself), or within a specific application container.
- Namespaces in a CDB
  In a CDB, the namespace for every object is scoped to its container.

## Principles of Commonality

In a CDB, a phenomenon can be common within either the system container (the CDB itself), or within a specific application container.

For example, if you create a common user account while connected to `CDB$ROOT`, then this user account is common to all PDBs and application roots in the CDB. If you create an application common user account while connected to an application root, however, then this user account is common only to the PDBs in this application container.

Within the context of `CDB$ROOT` or an application root, the principles of commonality are as follows:

- A common phenomenon is the same in every existing and future container.

  Therefore, a common user defined in the CDB root has the same identity in every PDB plugged in to the CDB root; a common user defined in an application root has the same identity in every application PDB plugged in to this application root. In contrast, a local phenomenon is scoped to exactly one existing container.
- Only a common user can alter the existence of common phenomena.

  More precisely, only a common user logged in to either the CDB root or an application root can create, destroy, or modify attributes of a user, role, or object that is common to the current container.

## Namespaces in a CDB

In a CDB, the namespace for every object is scoped to its container.

The following principles summarize the scoping rules:

- From an application perspective, a PDB is a separate database that is distinct from any other PDBs.
- Local phenomena are created within and restricted to a single container.

> **Note:**
>
> In this topic, the word “phenomenon” means “user account, role, or database object.”

- Common phenomena are defined in a CDB root or application root, and exist in all PDBs that are or will be plugged into this root.

The preceding principles have implications for local and common phenomena.

### Local Phenomena

A local phenomenon must be uniquely named *within* a container, but not across all containers in the CDB. Identically named local phenomena in different containers are distinct. For example, local user `sh` in one PDB does not conflict with local user `sh` in another PDB.

### CDB$ROOT Common Phenomena

Common phenomena defined in `CDB$ROOT` exist in multiple containers and must be unique within each of these namespaces. For example, the CDB root includes predefined common users such as `SYSTEM` and `SYS`. To ensure namespace separation, Oracle Database prevents creation of a `SYSTEM` user within another container.

To ensure namespace separation, the name of user-created common phenomena in the CDB root must begin with the value specified by the `COMMON_USER_PREFIX` initialization parameter. The default prefix is `c##` or `C##`. The names of all *other* user-created phenomena must *not* begin with `c##` or `C##`. For example, you cannot create a local user in `hrpdb` named `c##hr`, nor can you create a common user in the CDB root named `hr`.

### Application Common Phenomena

Within an application container, names for local and application common phenomena must not conflict.

- Application common users and roles

  The same principles apply to application common users as to CDB common users. The difference is that for CDB common users, the default value for the common user prefix is `c##` or `C##`, whereas in application root the default value for the common user prefix is the empty string.

  The multitenant architecture assumes that you create application PDBs from an application root, or convert a single-tenant application to a multitenant application.

- Application common objects

  The multitenant architecture assumes that you create application common objects in the application root. Later, you add data locally within the application PDBs. However, Oracle Database supports creation of *local* tables within an application PDB. In this case, the local tables reside in the same namespace as application common objects within the application PDB.

> **See Also:**
>
> *Oracle Database Security Guide* to learn more about common users and roles

## Creation of Application Common Objects

To create common objects, connect to an application root, and then execute a `CREATE` statement that specifies a sharing attribute.

You can only create or change application common objects as part of an application installation, upgrade, or patch. You can specify sharing in the following ways:

- `DEFAULT_SHARING` initialization parameter

  The setting is the default sharing attribute for all database objects of a supported type created in the root.

- `SHARING` clause

  You specify this clause in the `CREATE` statement itself. When a `SHARING` clause is included in a SQL statement, it takes precedence over the value specified in the `DEFAULT_SHARING` initialization parameter. Possible values are `METADATA`, `DATA`, `EXTENDED DATA`, and `NONE`.

The following table shows the types of application common objects, and where the data and metadata is stored.

**Table 3-1 Application Common Objects**

| Object Type | SHARING Value | Metadata Storage | Data Storage |
|---|---|---|---|
| Data-Linked | `DATA` | Application root | Application root |
| Extended Data-Linked | `EXTENDED DATA` | Application root | Application root and application PDB |
| Metadata-Linked | `METADATA` | Application root | Application PDB |

> **See Also:**
>
> *Oracle Database Security Guide* to learn how to manage privileges for common objects

## Metadata-Linked Application Common Objects

A **metadata link** is a dictionary object that supports referring to, and granting privileges on, common metadata shared by all PDBs in the application container.

Specifying the `METADATA` value in either the `SHARING` clause or the `DEFAULT_SHARING` initialization parameter specifies a link to an object's metadata, called a metadata-linked common object. The metadata for the object is stored once in the application root.

Tables, views, and code objects (such as PL/SQL procedures) can share metadata. In this context, "metadata" includes column definitions, constraints, triggers, and code. For example, if

sales_mlt is a metadata-linked common table, then all application PDBs access the *same* definition of this table, which is stored in the application root, by means of a metadata link. The rows in sales_mlt are different in every application PDB, but the column definitions are the same.

Typically, most objects in an application will be metadata-linked. Thus, you need only maintain one master application definition. This approach centralizes management of the application in multiple application PDBs.

**Example 3-2 Creating a Metadata-Linked Common Object**

In this example, the SYSTEM user logs in to the saas_sales_ac application container. SYSTEM installs an application named saas_sales_app at version 1.0. This application creates a common user account named saas_sales_adm. The schema contains a metadata-linked common table named sales_mlt.

```
-- Begin the install of saas_sales_app
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app BEGIN INSTALL '1.0';

-- Create the tablespace for the app
CREATE TABLESPACE saas_sales_tbs DATAFILE SIZE 100M AUTOEXTEND ON NEXT 10M
MAXSIZE 200M;

-- Create the user account saas_sales_adm, which will own the app
CREATE USER saas_sales_adm IDENTIFIED BY ****** CONTAINER=ALL;

-- Grant necessary privileges to this user account
GRANT CREATE SESSION, DBA TO saas_sales_adm;

-- Makes the tablespace that you just created the default for saas_sales_adm
ALTER USER saas_sales_adm DEFAULT TABLESPACE saas_sales_tbs;

-- Now connect as the application owner
CONNECT saas_sales_adm/******@saas_sales_ac

-- Create a metadata-linked table
CREATE TABLE saas_sales_adm.sales_mlt SHARING=METADATA
(YEAR       NUMBER(4),
 REGION     VARCHAR2(10),
 QUARTER    VARCHAR2(4),
 REVENUE    NUMBER);

-- End the application installation
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app END INSTALL '1.0';
```

You can use the ALTER PLUGGABLE DATABASE APPLICATION ... SYNC statement to synchronize the application PDBs to use the same master application definition. In this way, every application PDB has a metadata link to the saas_sales_adm.sales_mlt common table. The middle-tier code that updates sales_mlt within the PDB named cust1_pdb adds rows to this table in cust1_pdb, whereas the middle-tier code that updates sales_mlt in cust2_pdb adds rows to the copy of this table in cust2_pdb. Only the table metadata, which is stored in the application root, is shared.

> **Note:**
>
> *Oracle Database Security Guide* to learn more about how commonly granted object privileges work

- Metadata Links
  For metadata-linked application common objects, the metadata for the object is stored once in the application root. A metadata link is a dictionary object whose object type is the same as the metadata it is sharing.

## Metadata Links

For metadata-linked application common objects, the metadata for the object is stored once in the application root. A metadata link is a dictionary object whose object type is the same as the metadata it is sharing.

The description of a metadata link is stored in the data dictionary of the PDB in which it is created. A metadata link must be owned by an application common user. You can only use metadata links to share metadata of common objects owned by their creator in the CDB root or an application root.

Unlike a data link, a metadata link depends *only* on common data. For example, if an application contains the local tables `dow_close_lt` and `nasdaq_close_lt` in the application root, then a common user cannot create metadata links to these objects. However, an application common table named `sales_mlt` may be metadata-linked.

If a privileged common user changes the metadata for `sales_mlt`, for example, adds a column to the table, then this change propagates to the metadata links. Application PDB users may not change the metadata in the metadata link. For example, a DBA who manages the application PDB named `cust1_pdb` cannot add a column to `sales_mlt` in this PDB only: such metadata changes can be made only in the application root.

# Data-Linked Application Common Objects

A **data-linked object** is an object whose metadata and data reside in an application root, and are accessible from all application PDBs in this application container.

Specifying the `DATA` value in either the `SHARING` clause or the `DEFAULT_SHARING` initialization parameter specifies a link to a common object, called a data-linked common object. Dimension tables in a data warehouse are often good candidates for data-linked common tables.

A data link is a dictionary object that functions much like a synonym. For example, if `countries` is an application common table, then all application PDBs access the *same* copy of this table by means of a data link. If a row is added to this table, then this row is visible in all application PDBs.

A data link must be owned by an application common user. The link inherits the object type from the object to which it is pointing. The description of a data link is stored in the dictionary of the PDB in which it is created. For example, if an application container contains 10 application PDBs, and if every PDB contains a link to the `countries` application common table, then all 10 PDBs contain dictionary definitions for this link.

**Example 3-3 Creating a Data-Linked Object**

In this example, `SYSTEM` connects to the `saas_sales_ac` application container. `SYSTEM` upgrades the application named `saas_sales_app` from version `1.0` to `2.0`. This application upgrade logs in to the container as common user `saas_sales_adm`, creates a data-linked table named `countries_dlt`, and then inserts rows into it.

```
-- Begin an upgrade of the application
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app BEGIN UPGRADE '1.0' to
'2.0';

-- Connect as application owner to application root
CONNECT saas_sales_adm/manager@saas_sales_ac

-- Create data-linked table named countries_dlt
CREATE TABLE countries_dlt SHARING=DATA
(country_id   NUMBER,
 country_name VARCHAR2(20));

-- Insert records into countries_dlt
INSERT INTO countries_dlt VALUES(1, 'USA');
INSERT INTO countries_dlt VALUES(44, 'UK');
INSERT INTO countries_dlt VALUES(86, 'China');
INSERT INTO countries_dlt VALUES(91, 'India');

-- End application upgrade
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app END UPGRADE TO '2.0';
```

Use the `ALTER PLUGGABLE DATABASE APPLICATION ... SYNC` statement to synchronize application PDBs with the application root. In this way, every synchronized application PDB has a data link to the `saas_sales_adm.countries_dlt` data-linked table.

## Extended Data-Linked Application Objects

An **extended data-linked object** is a hybrid of a data-linked object and metadata-linked object.

In an extended data-linked object, the data stored in the application root is common to all application PDBs, and all PDBs can access this data. However, each application PDB can create its own, PDB-specific data while sharing the common data in application root. Thus, the PDBs supplement the common data with their own data.

For example, a sales application might support several application PDBs. All application PDBs need the postal codes for the United States. In this case, you might create a `zipcodes_edt` extended data-linked table in the application root. The application root stores the United States postal codes, so all application PDBs can access them. However, one application PDB requires the postal codes for the United States and Canada. This application PDB can store the postal codes for Canada in the extended data-linked object in the application PDB instead of in the application root.

Create an extended data-linked object by connecting to the application root and specifying the `SHARING=EXTENDED DATA` keyword in the `CREATE` statement.

**Example 3-4 Creating an Extended-Data Object**

In this example, `SYSTEM` connects to the `saas_sales_ac` application container, and then upgrades the application named `saas_sales_app` (created in "Example 3-2") from version `2.0` to `3.0`. This application logs in to the container as common user `saas_sales_adm`, creates an extended data-linked table named `zipcodes_edt`, and then inserts rows into it.

```
-- Begin an upgrade of the app
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app BEGIN UPGRADE '2.0' to
'3.0';

-- Connect as app owner to app root
CONNECT saas_sales_adm/manager@saas_sales_ac

-- Create a common-data table named zipcodes_edt
CREATE TABLE zipcodes_edt SHARING=EXTENDED DATA
(code       VARCHAR2(5),
 country_id NUMBER,
 region     VARCHAR2(10));

-- Load rows into zipcodes_edt
INSERT INTO zipcodes_edt VALUES ('08820','1','East');
INSERT INTO zipcodes_edt VALUES ('10005','1','East');
INSERT INTO zipcodes_edt VALUES ('44332','1','North');
INSERT INTO zipcodes_edt VALUES ('94065','1','West');
INSERT INTO zipcodes_edt VALUES ('73301','1','South');
COMMIT;

-- End app upgrade
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app END UPGRADE TO '3.0';
```

Use the `ALTER PLUGGABLE DATABASE APPLICATION ... SYNC` statement to synchronize application PDBs with the application. In this way, every synchronized application PDB has a data link to the `saas_sales_adm.zipcodes_edt` data-linked table. Applications that connect to these PDBs can see the postal codes that were inserted into `zipcodes_edt` during the application upgrade, but can also insert their own postal codes into this table.

# Container Maps

A **container map** enables a session connected to application root to issue SQL statements that are routed to the appropriate PDB, depending on the value of a predicate used in the SQL statement.

A map table specifies a column in a metadata-linked common table, and uses partitions to associate different application PDBs with different column values. In this way, container maps enable the partitioning of data at the PDB level when the data is not physically partitioned at the table level.

The key components for using container maps are:

- Metadata-linked table

  This table is intended to be queried using the container map. For example, you might create a metadata-linked table named `countries_mlt` that stores different data in each application PDB. In `amer_pdb`, the `countries_mlt.cname` column stores North American

country names; in euro_pdb, the countries_mlt.cname column stores European country names; and in asia_pdb, the countries_mlt.cname column stores Asian country names.

- Map table

  In the application root, you create a single-column map table partitioned by list, hash, or range. The map table enables the metadata-linked table to be queried using the partitioning strategy that is enabled by the container map. The names of the partitions in the map object table must match the names of the application PDBs in the application container.

  For example, the map table named pdb_map_tbl may partition by list on the cname column. The partitions named amer_pdb, euro_pdb, and asia_pdb correspond to the names of the application PDBs. The values in each partition are the names of the countries, for example, PARTITION amer_pdb VALUES ('US','MEXICO','CANADA').

  Starting in Oracle Database 18c, for a CONTAINERS() query to use a map, the partitioning column in the map table does not need to match a column in the metadata-linked table. Assume that the table sh.sales is enabled for the container map pdb_map_tbl, and cname is the partitioning column for the map table. Even though sh.sales does *not* include a cname column, the map table routes the following query to the appropriate PDB: SELECT * FROM CONTAINERS(sh.sales) WHERE cname = 'US' ORDER BY time_id.

- Container map

  A container map is a database property that specifies a map table. To set the property, you connect to the application root and execute the ALTER PLUGGABLE DATABASE SET CONTAINER_MAP=*map_table* statement, where *map_table* is the name of the map table.

**Example 3-5 Creating a Metadata-Linked Table, Map Table, and Container Map: Part 1**

In this example, you log in as an application administrator to the application root. Assume that an application container has three application PDBs: amer_pdb, euro_pdb, and asia_pdb. Each application PDB stores country names for a different region. A metadata-linked table named oe.countries_mlt has a cname column that stores the country name. For this partitioning strategy, you use partition by list to create a map object named salesadm.pdb_map_tbl that creates a partition for each region. The country name determines the region.

```
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app BEGIN INSTALL '1.0';

-- Create the metadata-linked table.
CREATE TABLE oe.countries_mlt SHARING=METADATA (
  region    VARCHAR2(30),
  cname     VARCHAR2(30));

-- Create the partitioned map table, which is list partitioned on the
-- cname column. The names of the partitions are the names of the
-- application PDBs.
CREATE TABLE salesadm.pdb_map_tbl (cname VARCHAR2(30) NOT NULL)
  PARTITION BY LIST (cname) (
    PARTITION amer_pdb VALUES ('US','MEXICO','CANADA'),
    PARTITION euro_pdb VALUES ('UK','FRANCE','GERMANY'),
    PARTITION asia_pdb VALUES ('INDIA','CHINA','JAPAN'));

-- Set the CONTAINER_MAP database property to the map object.
ALTER PLUGGABLE DATABASE SET CONTAINER_MAP='salesadm.pdb_map_tbl';

-- Enable the container map for the metadata-linked table to be queried.
ALTER TABLE oe.countries_mlt ENABLE CONTAINER_MAP;
```

```
-- Ensure that the table to be queried is enabled for the
-- CONTAINERS clause.
ALTER TABLE oe.countries_mlt ENABLE CONTAINERS_DEFAULT;

-- End the application installation.
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app END INSTALL '1.0';
```

> **Note:**
>
> Although you create container maps using partitioning syntax, the database does not use partitioning functionality. Defining a container map does not require Oracle Partitioning.

In the preceding script, the `ALTER TABLE oe.countries_mlt ENABLE CONTAINERS_DEFAULT` statement specifies that queries and DML statements issued in the application root must use the `CONTAINERS()` clause by default for the database object.

**Example 3-6 Synchronizing the Application, and Adding Data: Part 2**

This example continues from the previous example. While connected to the application root, you switch the current container to each PDB in turn, synchronize the `saas_sales_app` application, and then add PDB-specific data to the `oe.countries_mlt` table.

```
ALTER SESSION SET CONTAINER=amer_pdb;
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app SYNC;
INSERT INTO oe.countries_mlt VALUES ('AMER','US');
INSERT INTO oe.countries_mlt VALUES ('AMER','MEXICO');
INSERT INTO oe.countries_mlt VALUES ('AMER','CANADA');
COMMIT;

ALTER SESSION SET CONTAINER=euro_pdb;
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app SYNC;
INSERT INTO oe.countries_mlt VALUES ('EURO','UK');
INSERT INTO oe.countries_mlt VALUES ('EURO','FRANCE');
INSERT INTO oe.countries_mlt VALUES ('EURO','GERMANY');
COMMIT;

ALTER SESSION SET CONTAINER=asia_pdb;
ALTER PLUGGABLE DATABASE APPLICATION saas_sales_app SYNC;
INSERT INTO oe.countries_mlt VALUES ('ASIA','INDIA');
INSERT INTO oe.countries_mlt VALUES ('ASIA','CHINA');
INSERT INTO oe.countries_mlt VALUES ('ASIA','JAPAN');
COMMIT;
```

**Example 3-7 Querying the Metadata-Linked Table: Part 3**

This example continues from the previous example. You connect to the application root, and then query `oe.countries_mlt` multiple times, specifying different countries in the `WHERE` clause. The query returns the correct value from the `oe.countries_mlt.region` column.

```
ALTER SESSION SET CONTAINER=saas_sales_ac;
```

```
SELECT region FROM oe.countries_mlt WHERE cname='MEXICO';

REGION
------
AMER

SELECT region FROM oe.countries_mlt WHERE cname='GERMANY';

REGION
------
EURO

SELECT region FROM oe.countries_mlt WHERE cname='JAPAN';

REGION
------
ASIA
```

# Cross-Container Operations

A **cross-container operation** is a DDL or DML statement that affects multiple containers at once.

Only a common user connected to either the CDB root or an application root can perform cross-container operations. A cross-container operation can affect:

- The CDB itself
- Multiple containers within a CDB
- Multiple phenomena such as common users or common roles that are represented in multiple containers
- A container to which the user issuing the DDL or DML statement is currently not connected

Examples of cross-container DDL operations include user `SYSTEM` granting a privilege commonly to another common user, and an `ALTER DATABASE . . . RECOVER` statement that applies to the entire CDB.

When you are connected to either the CDB root or an application root, you can execute a single DML statement to modify tables or views in multiple PDBs within the container. The database infers the target PDBs from the value of the `CON_ID` column specified in the DML statement. If no `CON_ID` is specified, then the database uses the `CONTAINERS_DEFAULT_TARGET` property specified by the `ALTER PLUGGABLE DATABASE CONTAINERS DEFAULT TARGET` statement.

**Example 3-8 Updating Multiple PDBs in a Single DML Statement**

In this example, your goal is to set the `country_name` column to the value `USA` in the `sh.sales` table. This table exists in two separate PDBs, with container IDs of `7` and `8`. Both PDBs are in the application container named `saas_sales_ac`. You can connect to the application root as an administrator, and make the update as follows:

```
CONNECT sales_admin@saas_sales_ac
Password: *******

UPDATE CONTAINERS(sh.sales) sal
```

```
SET sal.country_name = 'USA'
WHERE sal.CON_ID IN (7,8);
```

In the preceding `UPDATE` statement, `sal` is an alias for `CONTAINERS(sh.sales)`.

> **See Also:**
>
> "Common User Accounts"

# Part II
# Oracle Relational Data Structures

This part describes the basic data structures of a database, including data integrity rules, and the structures that store metadata.

This part contains the following chapters:


- Tables and Table Clusters
- Indexes and Index-Organized Tables
- Partitions, Views, and Other Schema Objects
- Data Integrity
- Data Dictionary and Dynamic Performance Views


- Tables and Table Clusters
  This chapter provides an introduction to schema objects and discusses tables, which are the most common types of schema objects.
- Indexes and Index-Organized Tables
  Indexes are schema objects that can speed access to table rows. Index-organized tables are tables stored in an index structure.
- Partitions, Views, and Other Schema Objects
  Although tables and indexes are the most important and commonly used schema objects, the database supports many other types of schema objects, the most common of which are discussed in this chapter.
- Data Integrity
  This chapter explains how integrity constraints enforce the business rules associated with a database and prevent the entry of invalid information into tables.
- Data Dictionary and Dynamic Performance Views
  The central set of read-only reference tables and views of each Oracle database is known collectively as the **data dictionary**. The **dynamic performance views** are special views that are continuously updated while a database is open and in use.

4

# Tables and Table Clusters

This chapter provides an introduction to schema objects and discusses tables, which are the most common types of schema objects.

This chapter contains the following sections:


- Introduction to Schema Objects
- Overview of Tables
- Overview of Table Clusters
- Overview of Attribute-Clustered Tables
- Overview of Temporary Tables
- Overview of External Tables
- Overview of Blockchain Tables
- Overview of Object Tables


- Introduction to Schema Objects
  A database **schema** is a logical container for data structures, called schema objects. Examples of schema objects are tables and indexes. You create and manipulate schema objects with SQL.
- Overview of Tables
  A **table** is the basic unit of data organization in an Oracle database.
- Overview of Table Clusters
  A **table cluster** is a group of tables that share common columns and store related data in the same blocks.
- Overview of Attribute-Clustered Tables
  An **attribute-clustered table** is a heap-organized table that stores data in close proximity on disk based on user-specified clustering directives. The directives specify columns in single or multiple tables.
- Overview of Temporary Tables
  A **temporary table** holds data that exists only for the duration of a transaction or session.
- Overview of External Tables
  An **external table** accesses data in external sources as if this data were in a table in the database.
- Overview of Blockchain Tables
  A **blockchain table** is an append-only table designed for centralized blockchain applications.
- Overview of Immutable Tables
  Immutable tables are read-only tables that prevent unauthorized data modifications by insiders and accidental data modifications resulting from human errors.
- Overview of Object Tables
  An **object table** is a special kind of table in which each row represents an object.

# Introduction to Schema Objects

A database **schema** is a logical container for data structures, called schema objects. Examples of schema objects are tables and indexes. You create and manipulate schema objects with SQL.

This section contains the following topics:


- About Common and Local User Accounts
- Schema Object Types
- Schema Object Storage
- Schema Object Dependencies
- Sample Schemas


- About Common and Local User Accounts
  A database user account has a password and specific database privileges.
- Common and Local Objects
  A **common object** is defined in either the CDB root or an application root, and can be referenced using metadata links or object links. A local object is every object that is not a common object.
- Schema Object Types
  Oracle SQL enables you to create and manipulate many other types of schema objects.
- Schema Object Storage
  Some schema objects store data in a type of logical storage structure called a **segment**. For example, a nonpartitioned heap-organized table or an index creates a segment.
- Schema Object Dependencies
  Some schema objects refer to other objects, creating a **schema object dependency**.
- Sample Schemas
  An Oracle database may include **sample schemas**, which are a set of interlinked schemas that enable Oracle documentation and Oracle instructional materials to illustrate common database tasks.

> **See Also:**
>
> *Oracle Database Security Guide* to learn more about users and privileges

## About Common and Local User Accounts

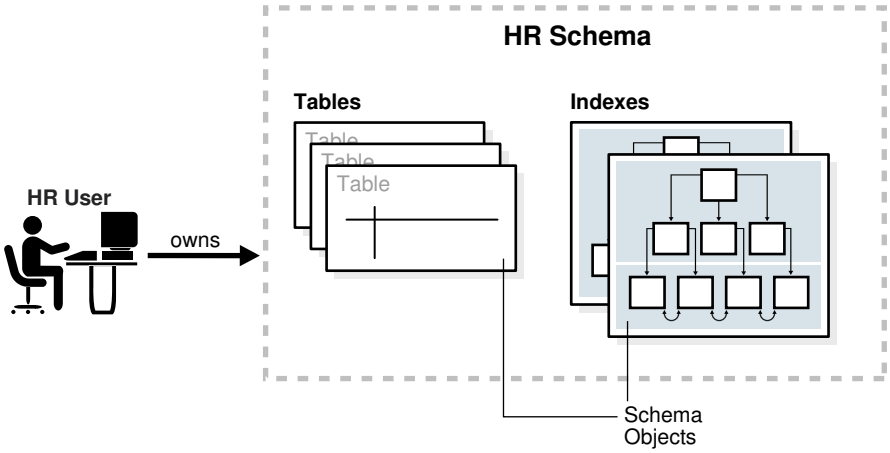
A database user account has a password and specific database privileges.

### User Accounts and Schemas

Each user account owns a single schema, which has the same name as the user. The schema contains the data for the user owning the schema. For example, the `hr` user account owns the `hr` schema, which contains schema objects such as the `employees` table. In a production database, the schema owner usually represents a database application rather than a person.

Within a schema, each schema object of a particular type has a unique name. For example, `hr.employees` refers to the table `employees` in the `hr` schema. The following figure depicts a schema owner named `hr` and schema objects within the `hr` schema.

**Figure 4-1 HR Schema**

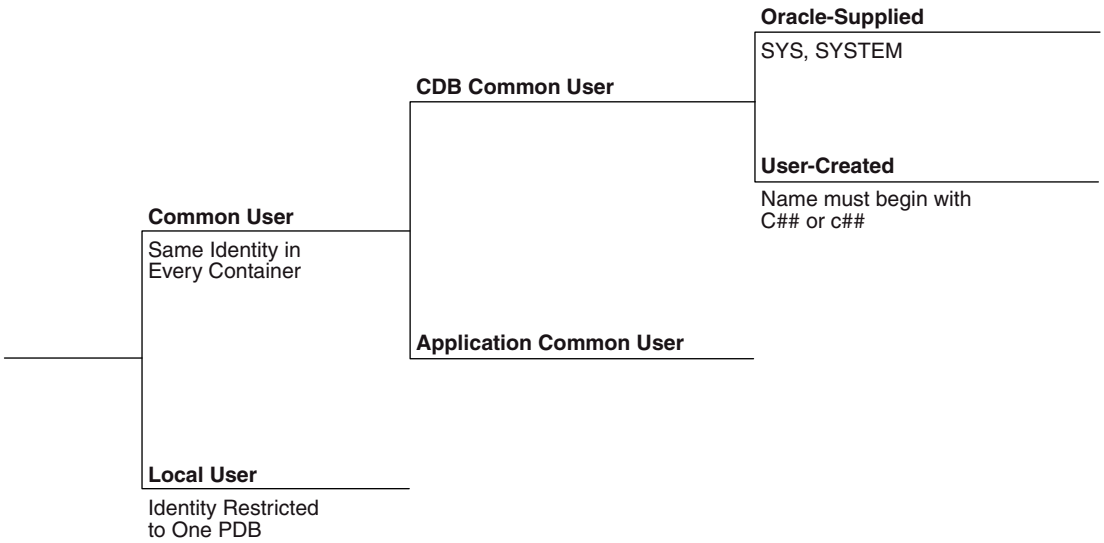



### Common and Local User Accounts

If a user account owns objects that define the database, then this user account is common. User accounts that are *not* Oracle-supplied are either local or common. A CDB common user is a common user that is created in the CDB root. An application common user is a user that is created in an application root, and is common only within this application container.

The following graphic shows the possible user account types in a CDB.

**Figure 4-2 User Accounts in a CDB**




A CDB common user can connect to *any* container in the CDB to which it has sufficient privileges. In contrast, an application common user can only connect to the application root in

which it was created, or a PDB that is plugged in to this application root, depending on its privileges.

- Common User Accounts
  Within the context of either the system container (CDB) or an application container, a **common user** is a database user that has the same identity in the root and in every existing and future PDB within this container.
- Local User Accounts
  A **local user** is a database user that is not common and can operate only within a single PDB.

## Common User Accounts

Within the context of either the system container (CDB) or an application container, a **common user** is a database user that has the same identity in the root and in every existing and future PDB within this container.

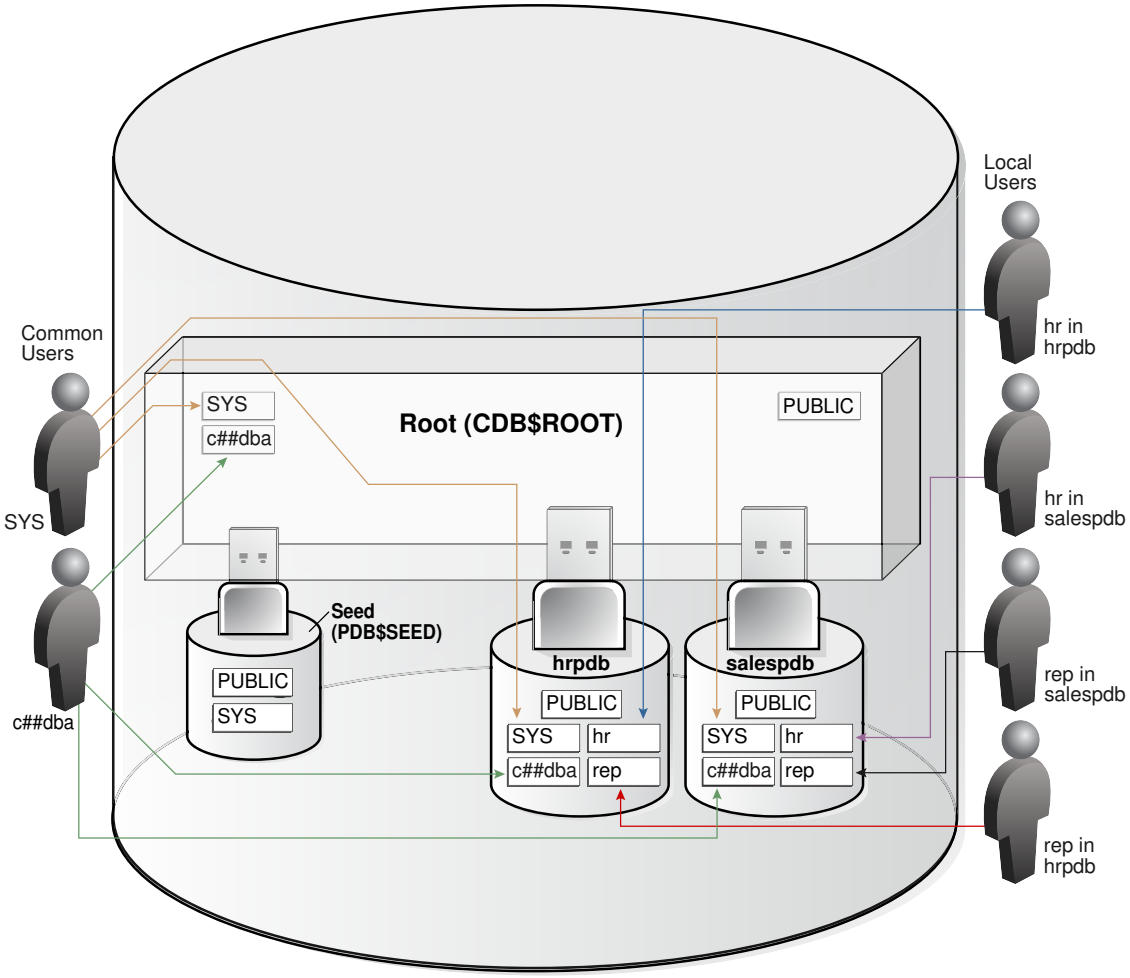
Every common user can connect to and perform operations within the root of its container, and within any PDB in which it has sufficient privileges. Some administrative tasks must be performed by a common user. Examples include creating a PDB and unplugging a PDB.

For example, `SYSTEM` is a CDB common user with DBA privileges. Thus, `SYSTEM` can connect to the CDB root and any PDB in the database. You might create a common user `saas_sales_admin` in the `saas_sales` application container. In this case, the `saas_sales_admin` user could only connect to the `saas_sales` application root or to an application PDB within the `saas_sales` application container.

Every common user is either Oracle-supplied or user-created. Examples of Oracle-supplied common users are `SYS` and `SYSTEM`. Every user-created common user is either a CDB common user, or an application common user.

The following figure shows sample users and schemas in two PDBs: `hrpdb` and `salespdb`. `SYS` and `c##dba` are CDB common users who have schemas in `CDB$ROOT`, `hrpdb`, and `salespdb`. Local users `hr` and `rep` exist in `hrpdb`. Local users `hr` and `rep` also exist in `salespdb`.

**Figure 4-3 Users and Schemas in a CDB**




Common users have the following characteristics:

- A common user can log in to any container (including `CDB$ROOT`) in which it has the `CREATE SESSION` privilege.
  A common user need not have the same privileges in every container. For example, the `c##dba` user may have the privilege to create a session in `hrpdb` and in the root, but not to create a session in `salespdb`. Because a common user with the appropriate privileges can switch between containers, a common user in the root can administer PDBs
- An application common user does not have the `CREATE SESSION` privilege in any container outside its own application container.
  Thus, an application common user is restricted to its own application container. For example, the application common user created in the `saas_sales` application can connect only to the application root and the PDBs in the `saas_sales` application container.
- The names of user-created CDB common users must follow the naming rules for other database users. Additionally, the names must begin with the characters specified by the `COMMON_USER_PREFIX` initialization parameter, which are `c##` or `C##` by default. Oracle-supplied common user names and user-created application common user names do not have this restriction.
  No local user name may begin with the characters `c##` or `C##`.

- Every common user is uniquely named across all PDBs within the container (either the system container or a specific application container) in which it was created.
  A CDB common user is defined in the CDB root, but must be able to connect to every PDB with the same identity. An application common user resides in the application root, and may connect to every application PDB in its container with the same identity.
- Characteristics of Common Users
  Every common user is either Oracle-supplied or user-created.
- SYS and SYSTEM Accounts
  All Oracle databases include default common user accounts with administrative privileges.

## Characteristics of Common Users

Every common user is either Oracle-supplied or user-created.

Common user accounts have the following characteristics:

- A common user can log in to any container (including `CDB$ROOT`) in which it has the `CREATE SESSION` privilege.

  A common user need not have the same privileges in every container. For example, the `c##dba` user may have the privilege to create a session in `hrpdb` and in the root, but *not* to create a session in `salespdb`. Because a common user with the appropriate privileges can switch between containers, a common user in the root can administer PDBs.
- An application common user does not have the `CREATE SESSION` privilege in any container outside its own application container.

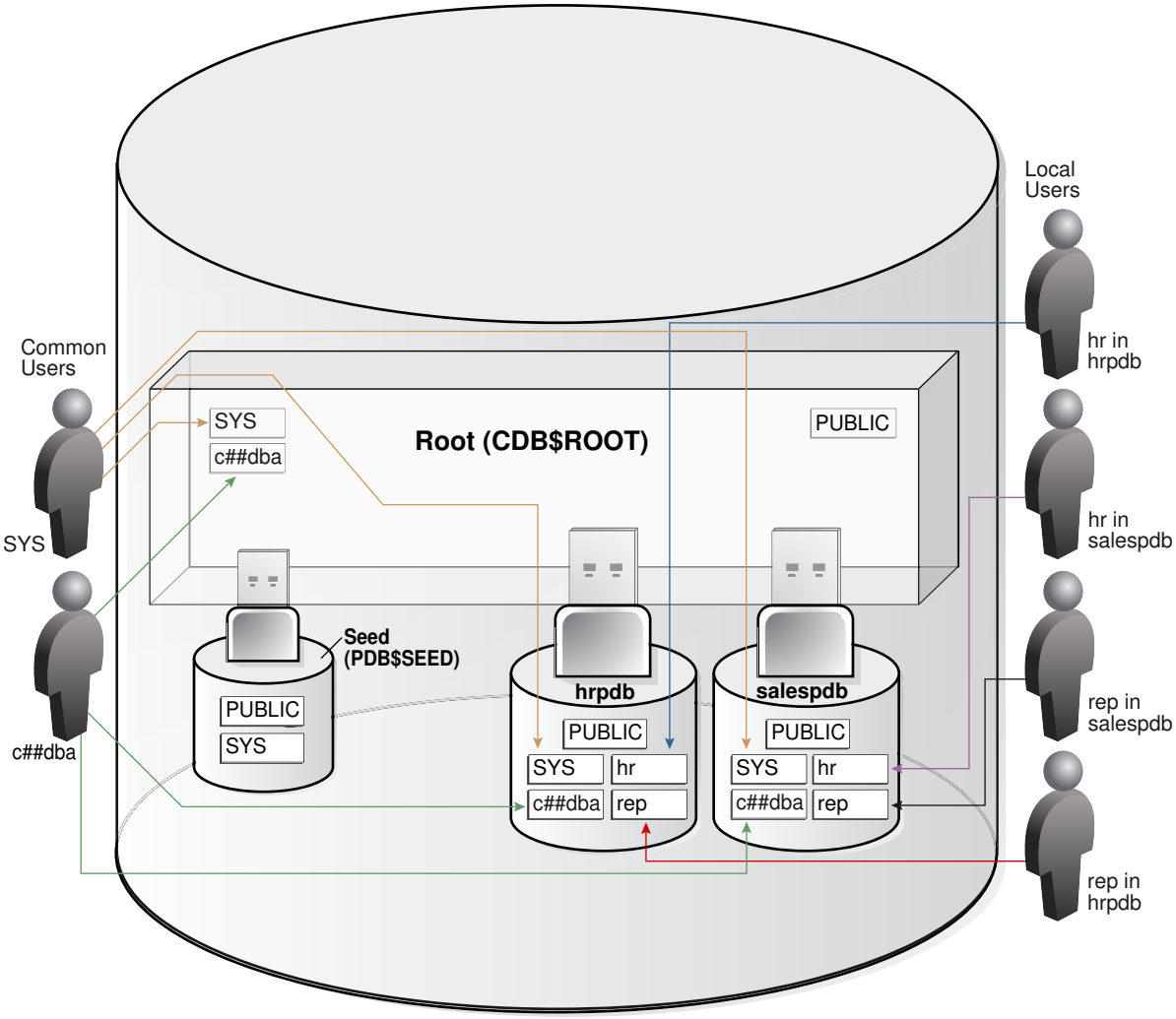
  Thus, an application common user is restricted to its own application container. For example, the application common user created in the `saas_sales` application can connect only to the application root and the PDBs in the `saas_sales` application container.
- The names of user-created CDB common users must follow the naming rules for other database users. Additionally, the names must begin with the characters specified by the `COMMON_USER_PREFIX` initialization parameter, which are `c##` or `C##` by default. Oracle-supplied common user names and user-created application common user names do not have this restriction.

  No local user name may begin with the characters `c##` or `C##`.
- Every common user is uniquely named across all PDBs within the container (either the system container or a specific application container) in which it was created.

  A CDB common user is defined in the CDB root, but must be able to connect to every PDB with the same identity. An application common user resides in the application root, and may connect to every application PDB *in its container* with the same identity.

The following figure shows sample users and schemas in two PDBs: `hrpdb` and `salespdb`. `SYS` and `c##dba` are CDB common users who have schemas in `CDB$ROOT`, `hrpdb`, and `salespdb`. Local users `hr` and `rep` exist in `hrpdb`. Local users `hr` and `rep` also exist in `salespdb`.

**Figure 4-4 Users and Schemas in a CDB**




> **See Also:**
>
> - *Oracle Database Security Guide* to learn about common user accounts
> - *Oracle Database Reference* to learn about `COMMON_USER_PREFIX`

## SYS and SYSTEM Accounts

All Oracle databases include default common user accounts with administrative privileges.

Administrative accounts are highly privileged and are intended only for DBAs authorized to perform tasks such as starting and stopping the database, managing memory and storage, creating and managing database users, and so on.

The `SYS` common user account is automatically created when a database is created. This account can perform all database administrative functions. The `SYS` schema stores the base

tables and views for the data dictionary. These base tables and views are critical for the operation of Oracle Database. Tables in the SYS schema are manipulated only by the database and must never be modified by any user.

The SYSTEM administrative account is also automatically created when a database is created. The SYSTEM schema stores additional tables and views that display administrative information, and internal tables and views used by various Oracle Database options and tools. Never use the SYSTEM schema to store tables of interest to nonadministrative users.

**See Also:**

- *Oracle Database Security Guide* to learn about user accounts
- *Oracle Database Administrator's Guide* to learn about SYS, SYSTEM, and other administrative accounts

## Local User Accounts

A **local user** is a database user that is not common and can operate only within a single PDB.

Local users have the following characteristics:

- A local user is specific to a PDB and may own a schema in this PDB.

  In the example shown in "Characteristics of Common Users", local user hr on hrpdb owns the hr schema. On salespdb, local user rep owns the rep schema, and local user hr owns the hr schema.

- A local user can administer a PDB, including opening and closing it.

  A common user with SYSDBA privileges can grant SYSDBA privileges to a local user. In this case, the privileged user remains local.

- A local user in one PDB cannot log in to another PDB or to the CDB root.

  For example, when local user hr connects to hrpdb, hr cannot access objects in the sh schema that reside in the salespdb database without using a database link. In the same way, when local user sh connects to the salespdb PDB, sh cannot access objects in the hr schema that resides in hrpdb without using a database link.

- The name of a local user must not begin with the characters c## or C##.

- The name of a local user must only be unique within its PDB.

  The user name and the PDB in which that user schema is contained determine a unique local user. "Characteristics of Common Users" shows that a local user and schema named rep exist on hrpdb. A completely independent local user and schema named rep exist on the salespdb PDB.

The following table describes a scenario involving the CDB in "Characteristics of Common Users". Each row describes an action that occurs after the action in the preceding row. Common user SYSTEM creates local users in two PDBs.

**Table 4-1 Local Users in a CDB**

| Operation | Description |
|---|---|
| `SQL> CONNECT SYSTEM@hrpdb`<br>`Enter password: ********`<br>`Connected.` | `SYSTEM` connects to the `hrpdb` container using the service name `hrpdb`. |
| `SQL> CREATE USER rep IDENTIFIED BY password;`<br><br>`User created.`<br><br>`SQL> GRANT CREATE SESSION TO rep;`<br><br>`Grant succeeded.` | `SYSTEM` now creates a local user `rep` and grants the `CREATE SESSION` privilege in this PDB to this user. The user is local because common users can only be created by a common user connected to the root. |
| `SQL> CONNECT rep@salespdb`<br>`Enter password: *******`<br>`ERROR:`<br>`ORA-01017: invalid username/password; logon denied` | The `rep` user, which is local to `hrpdb`, attempts to connect to `salespdb`. The attempt fails because `rep` does not exist in PDB `salespdb`. |
| `SQL> CONNECT SYSTEM@salespdb`<br>`Enter password: ********`<br>`Connected.` | `SYSTEM` connects to the `salespdb` container using the service name `salespdb`. |
| `SQL> CREATE USER rep IDENTIFIED BY password;`<br><br>`User created.`<br><br>`SQL> GRANT CREATE SESSION TO rep;`<br><br>`Grant succeeded.` | `SYSTEM` creates a local user `rep` in `salespdb` and grants the `CREATE SESSION` privilege in this PDB to this user. Because the name of a local user must only be unique within its PDB, a user named `rep` can exist in both `salespdb` and `hrpdb`. |
| `SQL> CONNECT rep@salespdb`<br>`Enter password: *******`<br>`Connected.` | The `rep` user successfully logs in to `salespdb`. |

**See Also:**

*Oracle Database Security Guide* to learn about local user accounts

## Common and Local Objects

A **common object** is defined in either the CDB root or an application root, and can be referenced using metadata links or object links. A local object is every object that is not a common object.

Database-supplied common objects are defined in `CDB$ROOT` and cannot be changed. Oracle Database does not support creation of common objects in `CDB$ROOT`.

You can create most schema objects—such as tables, views, PL/SQL and Java program units, sequences, and so on—as common objects in an application root. If the object exists in an application root, then it is called an **application common object**.

A local user can own a common object. Also, a common user can own a local object, but only when the object is not data-linked or metadata-linked, and is also neither a metadata link nor a data link.

> **See Also:**
>
> *Oracle Database Security Guide* to learn more about privilege management for common objects

## Schema Object Types

Oracle SQL enables you to create and manipulate many other types of schema objects.

The principal types of schema objects are shown in the following table.

**Table 4-2 Schema Objects**

| Object | Description | To Learn More |
|---|---|---|
| Table | A table stores data in rows. Tables are the most important schema objects in a relational database. | "Overview of Tables" |
| Indexes | Indexes are schema objects that contain an entry for each indexed row of the table or table cluster and provide direct, fast access to rows. Oracle Database supports several types of index. An index-organized table is a table in which the data is stored in an index structure. | "Indexes and Index-Organized Tables" |
| Partitions | Partitions are pieces of large tables and indexes. Each partition has its own name and may optionally have its own storage characteristics. | "Overview of Partitions" |
| Views | Views are customized presentations of data in one or more tables or other views. You can think of them as stored queries. Views do not actually contain data. | "Overview of Views" |

**Table 4-2 (Cont.) Schema Objects**

| Object | Description | To Learn More |
|---|---|---|
| Sequences | A sequence is a user-created object that can be shared by multiple users to generate integers. Typically, you use sequences to generate primary key values. | "Overview of Sequences" |
| Dimensions | A dimension defines a parent-child relationship between pairs of column sets, where all the columns of a column set must come from the same table. Dimensions are commonly used to categorize data such as customers, products, and time. | "Overview of Dimensions" |
| Synonyms | A synonym is an alias for another schema object. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary. | "Overview of Synonyms" |
| PL/SQL subprograms and packages | PL/SQL is the Oracle procedural extension of SQL. A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters. A PL/SQL package groups logically related PL/SQL types, variables, and subprograms. | "PL/SQL Subprograms " |

Other types of objects are also stored in the database and can be created and manipulated with SQL statements but are not contained in a schema. These objects include database user account, roles, contexts, and dictionary objects.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to manage schema objects
- *Oracle Database SQL Language Reference* for more about schema objects and database objects

## Schema Object Storage

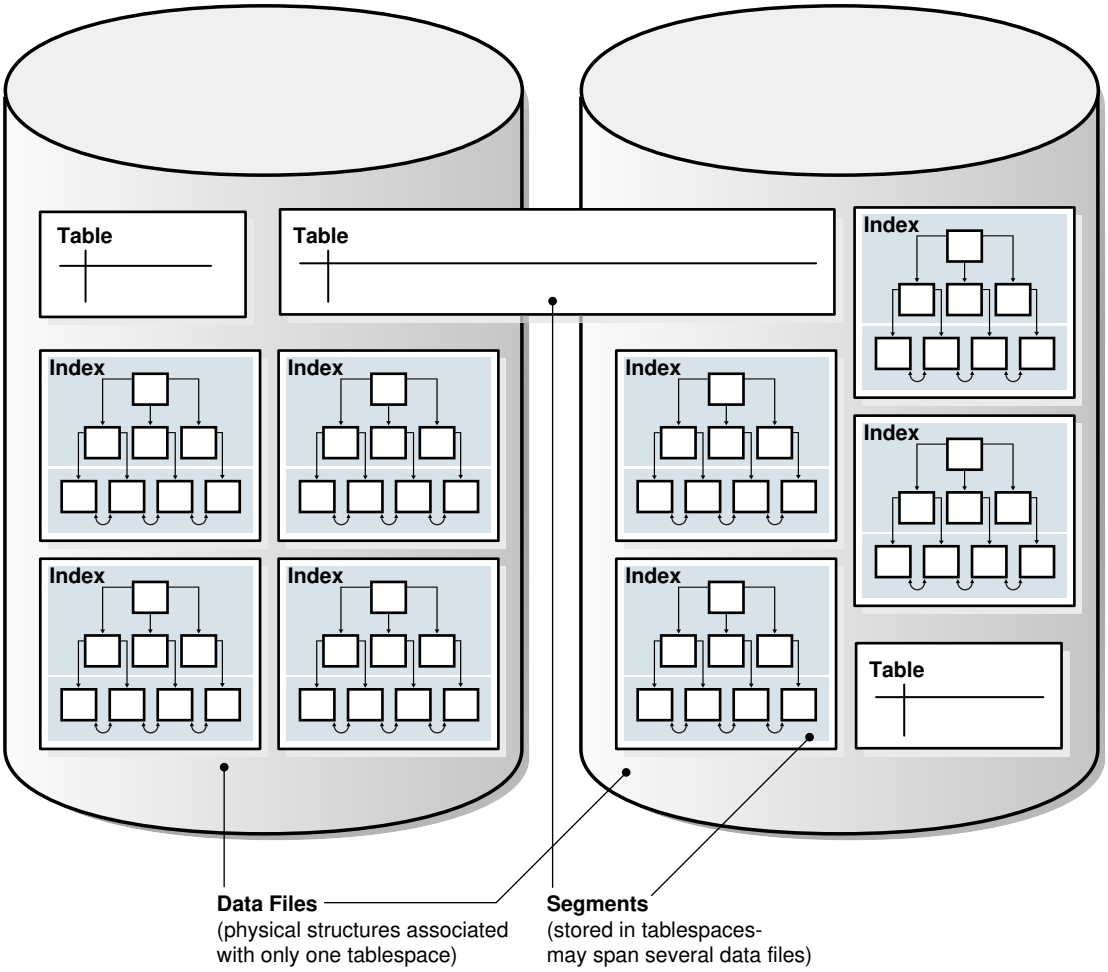
Some schema objects store data in a type of logical storage structure called a **segment**. For example, a nonpartitioned heap-organized table or an index creates a segment.

Other schema objects, such as views and sequences, consist of metadata only. This topic describes only schema objects that have segments.

Oracle Database stores a schema object logically within a tablespace. There is no relationship between schemas and tablespaces: a tablespace can contain objects from different schemas, and the objects for a schema can be contained in different tablespaces. The data of each object is physically contained in one or more data files.

The following figure shows a possible configuration of table and index segments, tablespaces, and data files. The data segment for one table spans two data files, which are both part of the same tablespace. A segment cannot span multiple tablespaces.

**Figure 4-5 Segments, Tablespaces, and Data Files**




> **See Also:**
>
> - "Logical Storage Structures" to learn about tablespaces and segments
> - *Oracle Database Administrator's Guide* to learn how to manage storage for schema objects

## Schema Object Dependencies

Some schema objects refer to other objects, creating a **schema object dependency**.

For example, a view contains a query that references tables or views, while a PL/SQL subprogram invokes other subprograms. If the definition of object A references object B, then A is a dependent object on B, and B is a referenced object for A.

Oracle Database provides an automatic mechanism to ensure that a dependent object is always up to date with respect to its referenced objects. When you create a dependent object, the database tracks dependencies between the dependent object and its referenced objects.

When a referenced object changes in a way that might affect a dependent object, the database marks the dependent object invalid. For example, if a user drops a table, no view based on the dropped table is usable.

An invalid dependent object must be recompiled against the new definition of a referenced object before the dependent object is usable. Recompilation occurs automatically when the invalid dependent object is referenced.

As an illustration of how schema objects can create dependencies, the following sample script creates a table `test_table` and then a procedure that queries this table:

```
CREATE TABLE test_table ( col1 INTEGER, col2 INTEGER );

CREATE OR REPLACE PROCEDURE test_proc
AS
BEGIN
 FOR x IN ( SELECT col1, col2 FROM test_table )
 LOOP
   -- process data
   NULL;
 END LOOP;
END;
/
```

The following query of the status of procedure `test_proc` shows that it is valid:

```
SQL> SELECT OBJECT_NAME, STATUS FROM USER_OBJECTS WHERE OBJECT_NAME =
'TEST_PROC';

OBJECT_NAME STATUS
----------- -------
TEST_PROC   VALID
```

After adding the `col3` column to `test_table`, the procedure is still valid because the procedure has no dependencies on this column:

```
SQL> ALTER TABLE test_table ADD col3 NUMBER;

Table altered.

SQL> SELECT OBJECT_NAME, STATUS FROM USER_OBJECTS WHERE OBJECT_NAME =
'TEST_PROC';

OBJECT_NAME STATUS
----------- -------
TEST_PROC   VALID
```

However, changing the data type of the `col1` column, which the `test_proc` procedure depends on, invalidates the procedure:

```
SQL> ALTER TABLE test_table MODIFY col1 VARCHAR2(20);

Table altered.
```

```
SQL> SELECT OBJECT_NAME, STATUS FROM USER_OBJECTS WHERE OBJECT_NAME =
'TEST_PROC';

OBJECT_NAME STATUS
----------- -------
TEST_PROC   INVALID
```

Running or recompiling the procedure makes it valid again, as shown in the following example:

```
SQL> EXECUTE test_proc

PL/SQL procedure successfully completed.

SQL> SELECT OBJECT_NAME, STATUS FROM USER_OBJECTS WHERE OBJECT_NAME =
'TEST_PROC';

OBJECT_NAME STATUS
----------- -------
TEST_PROC   VALID
```

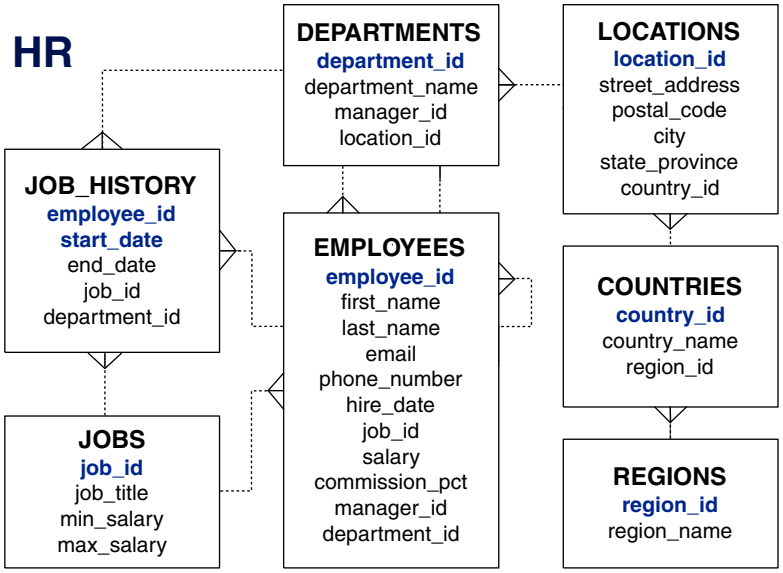
> **See Also:**
>
> *Oracle Database Administrator's Guide* and *Oracle Database Development Guide* to learn how to manage schema object dependencies

## Sample Schemas

An Oracle database may include **sample schemas**, which are a set of interlinked schemas that enable Oracle documentation and Oracle instructional materials to illustrate common database tasks.

The `hr` sample schema contains information about employees, departments and locations, work histories, and so on. The following illustration depicts an entity-relationship diagram of the tables in `hr`. Most examples in this manual use objects from this schema.

**Figure 4-6 HR Schema**




> **See Also:**
>
> *Oracle Database Sample Schemas* to learn how to install the sample schemas

# Overview of Tables

A **table** is the basic unit of data organization in an Oracle database.

A table describes an **entity**, which is something of significance about which information must be recorded. For example, an employee could be an entity.

Oracle Database tables fall into the following basic categories:

- Relational tables

  Relational tables have simple columns and are the most common table type. Example 4-1 shows a `CREATE TABLE` statement for a relational table.

- Object tables

  The columns correspond to the top-level attributes of an object type. See "Overview of Object Tables".

You can create a relational table with the following organizational characteristics:

- A heap-organized table does not store rows in any particular order. The `CREATE TABLE` statement creates a heap-organized table by default.
- An index-organized table orders rows according to the primary key values. For some applications, index-organized tables enhance performance and use disk space more efficiently. See "Overview of Index-Organized Tables".

- An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. See "Overview of External Tables".

A table is either permanent or temporary. A permanent table definition and data persist across sessions. A temporary table definition persists in the same way as a permanent table definition, but the data exists only for the duration of a transaction or session. Temporary tables are useful in applications where a result set must be held temporarily, perhaps because the result is constructed by running multiple operations.

This topic contains the following topics:

- Columns
- Rows
- Example: CREATE TABLE and ALTER TABLE Statements
- Oracle Data Types
- Integrity Constraints
- Table Storage
- Table Compression

- Columns
  A table definition includes a table name and set of columns.
- Rows
  A **row** is a collection of column information corresponding to a record in a table.
- Example: CREATE TABLE and ALTER TABLE Statements
  The Oracle SQL statement to create a table is `CREATE TABLE`.
- Oracle Data Types
  Each column has a **data type**, which is associated with a specific storage format, constraints, and valid range of values. The data type of a value associates a fixed set of properties with the value.
- Integrity Constraints
  An **integrity constraint** is a named rule that restrict the values for one or more columns in a table.
- Table Storage
  Oracle Database uses a **data segment** in a tablespace to hold table data.
- Table Compression
  The database can use **table compression** to reduce the amount of storage required for the table.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage tables

## Columns

A table definition includes a table name and set of columns.

A column identifies an attribute of the entity described by the table. For example, the column `employee_id` in the `employees` table refers to the employee ID attribute of an employee entity.

In general, you give each column a column name, a data type, and a width when you create a table. For example, the data type for `employee_id` is `NUMBER(6)`, indicating that this column can only contain numeric data up to 6 digits in width. The width can be predetermined by the data type, as with `DATE`.

- Virtual Columns
  A table can contain a **virtual column**, which unlike a nonvirtual column does not consume disk space.
- Invisible Columns
  An **invisible column** is a user-specified column whose values are only visible when the column is explicitly specified by name. You can add an invisible column to a table without affecting existing applications, and make the column visible if necessary.

## Virtual Columns

A table can contain a **virtual column**, which unlike a nonvirtual column does not consume disk space.

The database derives the values in a virtual column on demand by computing a set of user-specified expressions or functions. For example, the virtual column `income` could be a function of the `salary` and `commission_pct` columns.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage virtual columns

## Invisible Columns

An **invisible column** is a user-specified column whose values are only visible when the column is explicitly specified by name. You can add an invisible column to a table without affecting existing applications, and make the column visible if necessary.

In general, invisible columns help migrate and evolve online applications. A use case might be an application that queries a three-column table with a `SELECT *` statement. Adding a fourth column to the table would break the application, which expects three columns of data. Adding a fourth invisible column makes the application function normally. A developer can then alter the application to handle a fourth column, and make the column visible when the application goes live.

The following example creates a table `products` with an invisible column `count`, and then makes the invisible column visible:

```
CREATE TABLE products ( prod_id INT, count INT INVISIBLE );
ALTER TABLE products MODIFY ( count VISIBLE );
```

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to manage invisible columns
> - *Oracle Database SQL Language Reference* for more information about invisible columns

## Rows

A **row** is a collection of column information corresponding to a record in a table.

For example, a row in the `employees` table describes the attributes of a specific employee: employee ID, last name, first name, and so on. After you create a table, you can insert, query, delete, and update rows using SQL.

## Example: CREATE TABLE and ALTER TABLE Statements

The Oracle SQL statement to create a table is `CREATE TABLE`.

**Example 4-1 CREATE TABLE employees**

The following example shows the `CREATE TABLE` statement for the `employees` table in the `hr` sample schema. The statement specifies columns such as `employee_id`, `first_name`, and so on, specifying a data type such as `NUMBER` or `DATE` for each column.

```
CREATE TABLE employees
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
         CONSTRAINT     emp_last_name_nn  NOT NULL
    , email          VARCHAR2(25)
        CONSTRAINT     emp_email_nn  NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE
        CONSTRAINT     emp_hire_date_nn  NOT NULL
    , job_id         VARCHAR2(10)
        CONSTRAINT     emp_job_nn  NOT NULL
    , salary         NUMBER(8,2)
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , CONSTRAINT     emp_salary_min
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk
                     UNIQUE (email)
    ) ;
```

### Example 4-2 ALTER TABLE employees

The following example shows an `ALTER TABLE` statement that adds integrity constraints to the `employees` table. Integrity constraints enforce business rules and prevent the entry of invalid information into tables.

```
ALTER TABLE employees
ADD ( CONSTRAINT     emp_emp_id_pk
                       PRIMARY KEY (employee_id)
    , CONSTRAINT     emp_dept_fk
                       FOREIGN KEY (department_id)
                         REFERENCES departments
    , CONSTRAINT     emp_job_fk
                       FOREIGN KEY (job_id)
                         REFERENCES jobs (job_id)
    , CONSTRAINT     emp_manager_fk
                       FOREIGN KEY (manager_id)
                         REFERENCES employees
    ) ;
```

### Example 4-3 Rows in the employees Table

The following sample output shows 8 rows and 6 columns of the `hr.employees` table.

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME      SALARY COMMISSION_PCT DEPARTMENT_ID
----------- ----------- ------------- ------- -------------- -------------
        100 Steven      King            24000                          90
        101 Neena       Kochhar         17000                          90
        102 Lex         De Haan         17000                          90
        103 Alexander   Hunold           9000                          60
        107 Diana       Lorentz          4200                          60
        149 Eleni       Zlotkey         10500             .2           80
        174 Ellen       Abel            11000             .3           80
        178 Kimberely   Grant            7000            .15
```

The preceding output illustrates some of the following important characteristics of tables, columns, and rows:

- A row of the table describes the attributes of one employee: name, salary, department, and so on. For example, the first row in the output shows the record for the employee named Steven King.
- A column describes an attribute of the employee. In the example, the `employee_id` column is the primary key, which means that every employee is uniquely identified by employee ID. Any two employees are guaranteed not to have the same employee ID.
- A non-key column can contain rows with identical values. In the example, the salary value for employees 101 and 102 is the same: `17000`.
- A foreign key column refers to a primary or unique key in the same table or a different table. In this example, the value of `90` in `department_id` corresponds to the `department_id` column of the `departments` table.
- A field is the intersection of a row and column. It can contain only one value. For example, the field for the department ID of employee 103 contains the value `60`.
- A field can lack a value. In this case, the field is said to contain a null value. The value of the `commission_pct` column for employee 100 is null, whereas the value in the field for

employee 149 is `.2`. A column allows nulls unless a `NOT NULL` or primary key integrity constraint has been defined on this column, in which case no row can be inserted without a value for this column.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `CREATE TABLE` syntax and semantics

## Oracle Data Types

Each column has a **data type**, which is associated with a specific storage format, constraints, and valid range of values. The data type of a value associates a fixed set of properties with the value.

These properties cause Oracle Database to treat values of one data type differently from values of another. For example, you can multiply values of the `NUMBER` data type, but not values of the `RAW` data type.

When you create a table, you must specify a data type for each of its columns. Each value subsequently inserted in a column assumes the column data type.

Oracle Database provides several built-in data types. The most commonly used data types fall into the following categories:

- Character Data Types
- Numeric Data Types
- Datetime Data Types
- Rowid Data Types
- Format Models and Data Types

Other important categories of built-in types include raw, large objects (LOBs), and collections. PL/SQL has data types for constants and variables, which include `BOOLEAN`, reference types, composite types (records), and user-defined types.

- Character Data Types
  Character data types store alphanumeric data in strings. The most common character data type is `VARCHAR2`, which is the most efficient option for storing character data.
- Numeric Data Types
  The Oracle Database numeric data types store fixed and floating-point numbers, zero, and infinity. Some numeric types also store values that are the undefined result of an operation, which is known as "not a number" or `NaN`.
- Datetime Data Types
  The **datetime** data types are `DATE` and `TIMESTAMP`. Oracle Database provides comprehensive time zone support for time stamps.
- Rowid Data Types
  Every row stored in the database has an address– an internal representation to locate that row within the database. Oracle Database uses a `ROWID` data type to store the address (rowid) of every row in the database.

- Format Models and Data Types
  A **format model** is a character literal that describes the format of datetime or numeric data stored in a character string. A format model does not change the internal representation of the value in the database.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide*
> - *Oracle Database SQL Language Reference* to learn about built-in SQL data types
> - *Oracle Database PL/SQL Packages and Types Reference* to learn about PL/SQL data types
> - *Oracle Database Development Guide* to learn how to use the built-in data types

## Character Data Types

Character data types store alphanumeric data in strings. The most common character data type is `VARCHAR2`, which is the most efficient option for storing character data.

The byte values correspond to the character encoding scheme, generally called a character set. The database character set is established at database creation. Examples of character sets are 7-bit ASCII, EBCDIC, and Unicode UTF-8.

The length semantics of character data types are measurable in bytes or characters. The treatment of strings as a sequence of bytes is called byte semantics. This is the default for character data types. The treatment of strings as a sequence of characters is called character semantics. A character is a code point of the database character set.

- VARCHAR2 and CHAR Data Types
- NCHAR and NVARCHAR2 Data Types

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* to learn more about character sets
> - *Oracle Database 2 Day Developer's Guide* for a brief introduction to data types
> - *Oracle Database Development Guide* to learn how to choose a character data type

### VARCHAR2 and CHAR Data Types

The `VARCHAR2` data type stores variable-length character literals. A **literal** is a fixed data value. For example, `'LILA'`, `'St. George Island'`, and `'101'` are all character literals; `5001` is a numeric literal. Character literals are enclosed in single quotation marks so that the database can distinguish them from schema object names.

> **Note:**
>
> This manual uses the terms *text literal*, *character literal*, and *string* interchangeably.

When you create a table with a `VARCHAR2` column, you specify a maximum string length. In Example 4-1, the `last_name` column has a data type of `VARCHAR2(25)`, which means that any name stored in the column has a maximum of 25 bytes.

For each row, Oracle Database stores each value in the column as a variable-length field unless a value exceeds the maximum length, in which case the database returns an error. For example, in a single-byte character set, if you enter 10 characters for the `last_name` column value in a row, then the column in the row piece stores only 10 characters (10 bytes), not 25. Using `VARCHAR2` reduces space consumption.

In contrast to `VARCHAR2`, `CHAR` stores fixed-length character strings. When you create a table with a `CHAR` column, the column requires a string length. The default is 1 byte. The database uses blanks to pad the value to the specified length.

Oracle Database compares `VARCHAR2` values using nonpadded comparison semantics and compares `CHAR` values using blank-padded comparison semantics.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for details about blank-padded and nonpadded comparison semantics

## NCHAR and NVARCHAR2 Data Types

The `NCHAR` and `NVARCHAR2` data types store Unicode character data.
**Unicode** is a universal encoded character set that can store information in any language using a single character set. `NCHAR` stores fixed-length character strings that correspond to the national character set, whereas `NVARCHAR2` stores variable length character strings.

You specify a national character set when creating a database. The character set of `NCHAR` and `NVARCHAR2` data types must be either `AL16UTF16` or `UTF8`. Both character sets use Unicode encoding.

When you create a table with an `NCHAR` or `NVARCHAR2` column, the maximum size is always in character length semantics. Character length semantics is the default and only length semantics for `NCHAR` or `NVARCHAR2`.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information about Oracle's globalization support feature

## Numeric Data Types

The Oracle Database numeric data types store fixed and floating-point numbers, zero, and infinity. Some numeric types also store values that are the undefined result of an operation, which is known as "not a number" or `NaN`.

Oracle Database stores numeric data in variable-length format. Each value is stored in scientific notation, with 1 byte used to store the exponent. The database uses up to 20 bytes to store the mantissa, which is the part of a floating-point number that contains its significant digits. Oracle Database does not store leading and trailing zeros.

- NUMBER Data Type
  The `NUMBER` data type stores fixed and floating-point numbers. The database can store numbers of virtually any magnitude. This data is guaranteed to be portable among different operating systems running Oracle Database. The `NUMBER` data type is recommended for most cases in which you must store numeric data.
- Floating-Point Numbers
  Oracle Database provides two numeric data types exclusively for floating-point numbers: `BINARY_FLOAT` and `BINARY_DOUBLE`.

### NUMBER Data Type

The `NUMBER` data type stores fixed and floating-point numbers. The database can store numbers of virtually any magnitude. This data is guaranteed to be portable among different operating systems running Oracle Database. The `NUMBER` data type is recommended for most cases in which you must store numeric data.

You specify a fixed-point number in the form `NUMBER(`*`p`*`,`*`s`*`)`, where *`p`* and *`s`* refer to the following characteristics:

- Precision

  The precision specifies the total number of digits. If a precision is not specified, then the column stores the values exactly as provided by the application without any rounding.
- Scale

  The scale specifies the number of digits from the decimal point to the least significant digit. Positive scale counts digits to the right of the decimal point up to and including the least significant digit. Negative scale counts digits to the left of the decimal point up to but not including the least significant digit. If you specify a precision without a scale, as in `NUMBER(6)`, then the scale is 0.

In Example 4-1, the `salary` column is type `NUMBER(8,2)`, so the precision is 8 and the scale is 2. Thus, the database stores a salary of 100,000 as `100000.00`.

### Floating-Point Numbers

Oracle Database provides two numeric data types exclusively for floating-point numbers: `BINARY_FLOAT` and `BINARY_DOUBLE`.

These types support all of the basic functionality provided by the `NUMBER` data type. However, whereas `NUMBER` uses decimal precision, `BINARY_FLOAT` and `BINARY_DOUBLE` use binary precision, which enables faster arithmetic calculations and usually reduces storage requirements.

BINARY_FLOAT and BINARY_DOUBLE are approximate numeric data types. They store approximate representations of decimal values, rather than exact representations. For example, the value 0.1 cannot be exactly represented by either BINARY_DOUBLE or BINARY_FLOAT. They are frequently used for scientific computations. Their behavior is similar to the data types FLOAT and DOUBLE in Java and XMLSchema.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about precision, scale, and other characteristics of numeric types

## Datetime Data Types

The **datetime** data types are DATE and TIMESTAMP. Oracle Database provides comprehensive time zone support for time stamps.

- DATE Data Type
  The DATE data type stores date and time. Although datetimes can be represented in character or number data types, DATE has special associated properties.
- TIMESTAMP Data Type
  The TIMESTAMP data type is an extension of the DATE data type.

### DATE Data Type

The DATE data type stores date and time. Although datetimes can be represented in character or number data types, DATE has special associated properties.

The database stores dates internally as numbers. Dates are stored in fixed-length fields of 7 bytes each, corresponding to century, year, month, day, hour, minute, and second.

> **Note:**
>
> Dates fully support arithmetic operations, so you add to and subtract from dates just as you can with numbers.

The database displays dates according to the specified format model. A format model is a character literal that describes the format of a datetime in a character string. The standard date format is DD-MON-RR, which displays dates in the form 01-JAN-11.

RR is similar to YY (the last two digits of the year), but the century of the return value varies according to the specified two-digit year and the last two digits of the current year. Assume that in 1999 the database displays 01-JAN-11. If the date format uses RR, then 11 specifies 2011, whereas if the format uses YY, then 11 specifies 1911. You can change the default date format at both the database instance and session level.

Oracle Database stores time in 24-hour format—HH:MI:SS. If no time portion is entered, then by default the time in a date field is 00:00:00 A.M. In a time-only entry, the date portion defaults to the first day of the current month.

> **See Also:**
>
> - *Oracle Database Development Guide* for more information about centuries and date format masks
> - *Oracle Database SQL Language Reference* for information about datetime format codes
> - *Oracle Database Development Guide* to learn how to perform arithmetic operations with datetime data types

### TIMESTAMP Data Type

The `TIMESTAMP` data type is an extension of the `DATE` data type.

`TIMESTAMP` stores fractional seconds in addition to the information stored in the `DATE` data type. The `TIMESTAMP` data type is useful for storing precise time values, such as in applications that must track event order.

The `DATETIME` data types `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` are time-zone aware. When a user selects the data, the value is adjusted to the time zone of the user session. This data type is useful for collecting and evaluating date information across geographic regions.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for details about the syntax of creating and entering data in time stamp columns

## Rowid Data Types

Every row stored in the database has an address– an internal representation to locate that row within the database. Oracle Database uses a `ROWID` data type to store the address (rowid) of every row in the database.

Rowids fall into the following categories:

- Physical rowids store the addresses of rows in heap-organized tables, table clusters, and table and index partitions.
- Logical rowids store the addresses of rows in index-organized tables.
- Foreign rowids are identifiers in foreign tables, such as DB2 tables accessed through a gateway. They are not standard Oracle Database rowids.

A data type called the universal rowid, or urowid, supports all types of rowids.

- Use of Rowids
- ROWID Pseudocolumn
  Every table in an Oracle database has a pseudocolumn named `ROWID`.

## Use of Rowids

Oracle Database uses rowids internally for the construction of indexes.
A B-tree index, which is the most common type, contains an ordered list of keys divided into ranges. Each key is associated with a rowid that points to the associated row's address for fast access.

End users and application developers can also use rowids for several important functions:

- Rowids are a fast means of re-accessing a row if its `rowid` has previously been retrieved with a SELECT statement.
- Rowids provide the ability to see how a table is organized.

While you can create tables with columns defined using the `ROWID` data type, you should not store rowids with the intention of using them to access data at a later stage. Using rowids in this manner may yield unpredictable or incorrect results. 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.

## ROWID Pseudocolumn

Every table in an Oracle database has a pseudocolumn named `ROWID`.

A pseudocolumn behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. A pseudocolumn is also similar to a SQL function without arguments. Functions without arguments typically return the same value for every row in the result set, whereas pseudocolumns typically return a different value for each row.

Values of the `ROWID` pseudocolumn are strings representing the address of each row. These strings have the data type `ROWID`. This pseudocolumn is not evident when listing the structure of a table by executing `SELECT` or `DESCRIBE`, nor does the pseudocolumn consume space. However, the rowid of each row can be retrieved with a SQL query using the reserved word `ROWID` as a column name.

The following example queries the `ROWID` pseudocolumn to show the rowid of the row in the `employees` table for employee 100:

```
SQL> SELECT ROWID FROM employees WHERE employee_id = 100;

ROWID
------------------
AAAPecAAFAAAABSAAA
```

> **See Also:**
>
> - "Rowid Format"
> - *Oracle Database Development Guide* to learn how to identify rows by address
> - *Oracle Database SQL Language Reference* to learn about rowid types

## Format Models and Data Types

A **format model** is a character literal that describes the format of datetime or numeric data stored in a character string. A format model does not change the internal representation of the value in the database.

When you convert a character string into a date or number, a format model determines how the database interprets the string. In SQL, you can use a format model as an argument of the `TO_CHAR` and `TO_DATE` functions to format a value to be returned from the database or to format a value to be stored in the database.

The following statement selects the salaries of the employees in Department 80 and uses the `TO_CHAR` function to convert these salaries into character values with the format specified by the number format model `'$99,990.99'`:

```
SQL> SELECT last_name employee, TO_CHAR(salary, '$99,990.99') AS "SALARY"
  2  FROM   employees
  3  WHERE  department_id = 80 AND last_name = 'Russell';

EMPLOYEE                  SALARY
------------------------- -----------
Russell                    $14,000.00
```

The following example updates a hire date using the `TO_DATE` function with the format mask `'YYYY MM DD'` to convert the string `'1998 05 20'` to a `DATE` value:

```
SQL> UPDATE employees
  2  SET hire_date = TO_DATE('1998 05 20','YYYY MM DD')
  3  WHERE last_name = 'Hunold';
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about format models

## Integrity Constraints

An **integrity constraint** is a named rule that restrict the values for one or more columns in a table.

Data integrity rules prevent invalid data entry into tables. Also, constraints can prevent the deletion of a table when certain dependencies exist.

If a constraint is enabled, then the database checks data as it is entered or updated. Oracle Database prevents data that does not conform to the constraint from being entered. If a constraint is disabled, then Oracle Database allows data that does not conform to the constraint to enter the database.

In Example 4-1, the `CREATE TABLE` statement specifies `NOT NULL` constraints for the `last_name`, `email`, `hire_date`, and `job_id` columns. The constraint clauses identify the columns and the conditions of the constraint. These constraints ensure that the specified columns contain no

null values. For example, an attempt to insert a new employee without a job ID generates an error.

You can create a constraint when or after you create a table. You can temporarily disable constraints if needed. The database stores constraints in the data dictionary.

> **See Also:**
>
> - "Data Integrity" to learn about integrity constraints
> - "Overview of the Data Dictionary" to learn about the data dictionary
> - *Oracle Database SQL Language Reference* to learn about SQL constraint clauses

## Table Storage

Oracle Database uses a **data segment** in a tablespace to hold table data.

A segment contains extents made up of data blocks. The data segment for a table (or cluster data segment, for a table cluster) is located in either the default tablespace of the table owner or in a tablespace named in the `CREATE TABLE` statement.

- Table Organization
  By default, a table is organized as a heap, which means that the database places rows where they fit best rather than in a user-specified order. Thus, a heap-organized table is an unordered collection of rows.
- Row Storage
  The database stores rows in data blocks. Each row of a table containing data for less than 256 columns is contained in one or more row pieces.
- Rowids of Row Pieces
  A **rowid** is effectively a 10-byte physical address of a row.
- Storage of Null Values
  A **null** is the absence of a value in a column. Nulls indicate missing, unknown, or inapplicable data.

> **See Also:**
>
> "User Segments" to learn about the types of segments and how they are created

## Table Organization

By default, a table is organized as a heap, which means that the database places rows where they fit best rather than in a user-specified order. Thus, a heap-organized table is an unordered collection of rows.

> **Note:**
>
> Index-organized tables use a different principle of organization.

As users add rows, the database places the rows in the first available free space in the data segment. Rows are not guaranteed to be retrieved in the order in which they were inserted.

The `hr.departments` table is a heap-organized table. It has columns for department ID, name, manager ID, and location ID. As rows are inserted, the database stores them wherever they fit. A data block in the table segment might contain the unordered rows shown in the following example:

```
50,Shipping,121,1500
120,Treasury,,1700
70,Public Relations,204,2700
30,Purchasing,114,1700
130,Corporate Tax,,1700
10,Administration,200,1700
110,Accounting,205,1700
```

The column order is the same for all rows in a table. The database usually stores columns in the order in which they were listed in the `CREATE TABLE` statement, but this order is not guaranteed. For example, if a table has a column of type `LONG`, then Oracle Database always stores this column last in the row. Also, if you add a new column to a table, then the new column becomes the last column stored.

A table can contain a virtual column, which unlike normal columns does not consume space on disk. The database derives the values in a virtual column on demand by computing a set of user-specified expressions or functions. You can index virtual columns, collect statistics on them, and create integrity constraints. Thus, virtual columns are much like nonvirtual columns.

> **See Also:**
>
> - "Overview of Index-Organized Tables"
> - *Oracle Database SQL Language Reference* to learn about virtual columns

## Row Storage

The database stores rows in data blocks. Each row of a table containing data for less than 256 columns is contained in one or more row pieces.

If possible, Oracle Database stores each row as one row piece. However, if all of the row data cannot be inserted into a single data block, or if an update to an existing row causes the row to outgrow its data block, then the database stores the row using multiple row pieces.

Rows in a table cluster contain the same information as rows in nonclustered tables. Additionally, rows in a table cluster contain information that references the cluster key to which they belong.

> **See Also:**
>
> "Data Block Format" to learn about the components of a data block

## Rowids of Row Pieces

A **rowid** is effectively a 10-byte physical address of a row.

Every row in a heap-organized table has a rowid unique to this table that corresponds to the physical address of a row piece. For table clusters, rows in different tables that are in the same data block can have the same rowid.

Oracle Database uses rowids internally for the construction of indexes. For example, each key in a B-tree index is associated with a rowid that points to the address of the associated row for fast access. Physical rowids provide the fastest possible access to a table row, enabling the database to retrieve a row in as little as a single I/O.

> **See Also:**
>
> - "Rowid Format" to learn about the structure of a rowid
> - "Overview of B-Tree Indexes" to learn about the types and structure of B-tree indexes

## Storage of Null Values

A **null** is the absence of a value in a column. Nulls indicate missing, unknown, or inapplicable data.

Nulls are stored in the database if they fall between columns with data values. In these cases, they require 1 byte to store the length of the column (zero). Trailing nulls in a row require no storage because a new row header signals that the remaining columns in the previous row are null. For example, if the last three columns of a table are null, then no data is stored for these columns.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about null values

# Table Compression

The database can use **table compression** to reduce the amount of storage required for the table.

Compression saves disk space, reduces memory use in the database buffer cache, and in some cases speeds query execution. Table compression is transparent to database applications.

- Basic Table Compression and Advanced Row Compression
  Dictionary-based table compression provides good compression ratios for heap-organized tables.
- Hybrid Columnar Compression
  With Hybrid Columnar Compression, the database stores the same column for a group of rows together. The data block does not store data in row-major format, but uses a combination of both row and columnar methods.

## Basic Table Compression and Advanced Row Compression

Dictionary-based table compression provides good compression ratios for heap-organized tables.

Oracle Database supports the following types of dictionary-based table compression:

- Basic table compression

  This type of compression is intended for bulk load operations. The database does not compress data modified using conventional DML. You must use direct path INSERT operations, `ALTER TABLE . . . MOVE` operations, or online table redefinition to achieve basic table compression.

- Advanced row compression

  This type of compression is intended for OLTP applications and compresses data manipulated by any SQL operation. The database achieves a competitive compression ratio while enabling the application to perform DML in approximately the same amount of time as DML on an uncompressed table.

For the preceding types of compression, the database stores compressed rows in row major format. All columns of one row are stored together, followed by all columns of the next row, and so on. The database replaces duplicate values with a short reference to a symbol table stored at the beginning of the block. Thus, information that the database needs to re-create the uncompressed data is stored in the data block itself.

Compressed data blocks look much like normal data blocks. Most database features and functions that work on regular data blocks also work on compressed blocks.

You can declare compression at the tablespace, table, partition, or subpartition level. If specified at the tablespace level, then all tables created in the tablespace are compressed by default.

**Example 4-4 Table-Level Compression**

The following statement applies advanced row compression to the `orders` table:

```
ALTER TABLE oe.orders ROW STORE COMPRESS ADVANCED;
```

**Example 4-5 Partition-Level Compression**

The following example of a partial `CREATE TABLE` statement specifies advanced row compression for one partition and basic table compression for the other partition:

```
CREATE TABLE sales (
    prod_id     NUMBER     NOT NULL,
    cust_id     NUMBER     NOT NULL, ... )
 PCTFREE 5 NOLOGGING NOCOMPRESS
 PARTITION BY RANGE (time_id)
 ( partition sales_2013 VALUES LESS THAN(TO_DATE(...)) ROW STORE COMPRESS
```

```
BASIC,
   partition sales_2014 VALUES LESS THAN (MAXVALUE) ROW STORE COMPRESS
ADVANCED );
```

> **See Also:**
>
> - "Row Format" to learn how values are stored in a row
> - "Data Block Compression" to learn about the format of compressed data blocks
> - *Oracle Database Utilities* to learn about using SQL*Loader for direct path loads
> - *Oracle Database Administrator's Guide* and *Oracle Database Performance Tuning Guide* to learn about table compression

## Hybrid Columnar Compression

With Hybrid Columnar Compression, the database stores the same column for a group of rows together. The data block does not store data in row-major format, but uses a combination of both row and columnar methods.

Storing column data together, with the same data type and similar characteristics, dramatically increases the storage savings achieved from compression. The database compresses data manipulated by any SQL operation, although compression levels are higher for direct path loads. Database operations work transparently against compressed objects, so no application changes are required.

> **Note:**
>
> Hybrid Column Compression and In-Memory Column Store (IM column store) are closely related. The primary difference is that Hybrid Column Compression optimizes disk storage, whereas the IM column store optimizes memory storage.

- Types of Hybrid Columnar Compression
  If your underlying storage supports Hybrid Columnar Compression, then you can specify different types of compression, depending on your requirements.
- Compression Units
  Hybrid Columnar Compression uses a logical construct called a **compression unit** to store a set of rows.
- DML and Hybrid Columnar Compression
  Hybrid Columnar Compression has implications for row locking in different types of DML operations.

> **See Also:**
>
> "In-Memory Area" to learn more about the IM column store

## Types of Hybrid Columnar Compression

If your underlying storage supports Hybrid Columnar Compression, then you can specify different types of compression, depending on your requirements.

The compression options are:

- Warehouse compression

  This type of compression is optimized to save storage space, and is intended for data warehouse applications.

- Archive compression

  This type of compression is optimized for maximum compression levels, and is intended for historical data and data that does not change.

Hybrid Columnar Compression is optimized for data warehousing and decision support applications on Oracle Exadata storage. Oracle Exadata maximizes the performance of queries on tables that are compressed using Hybrid Columnar Compression, taking advantage of the processing power, memory, and Infiniband network bandwidth that are integral to the Oracle Exadata storage server.

Other Oracle storage systems support Hybrid Columnar Compression, and deliver the same space savings as on Oracle Exadata storage, but do not deliver the same level of query performance. For these storage systems, Hybrid Columnar Compression is ideal for in-database archiving of older data that is infrequently accessed.

## Compression Units

Hybrid Columnar Compression uses a logical construct called a **compression unit** to store a set of rows.

When you load data into a table, the database stores groups of rows in columnar format, with the values for each column stored and compressed together. After the database has compressed the column data for a set of rows, the database fits the data into the compression unit.

For example, you apply Hybrid Columnar Compression to a `daily_sales` table. At the end of every day, you populate the table with items and the number sold, with the item ID and date forming a composite primary key. The following table shows a subset of the rows in `daily_sales`.

**Table 4-3 Sample Table daily_sales**

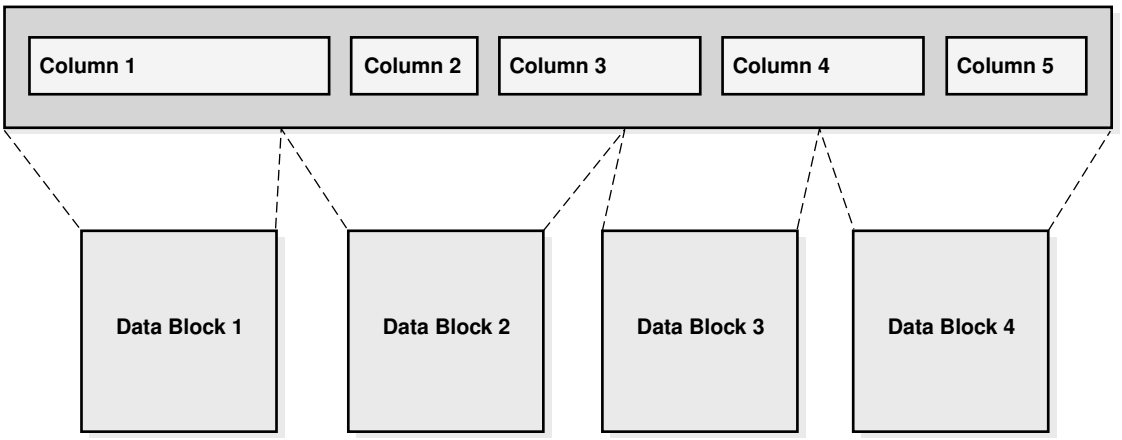
| Item_ID | Date | Num_Sold | Shipped_From | Restock |
|---|---|---|---|---|
| 1000 | 01-JUN-18 | 2 | WAREHOUSE1 | Y |
| 1001 | 01-JUN-18 | 0 | WAREHOUSE3 | N |
| 1002 | 01-JUN-18 | 1 | WAREHOUSE3 | N |
| 1003 | 01-JUN-14 | 0 | WAREHOUSE2 | N |
| 1004 | 01-JUN-18 | 2 | WAREHOUSE1 | N |
| 1005 | 01-JUN-18 | 1 | WAREHOUSE2 | N |

Assume that this subset of rows is stored in one compression unit. Hybrid Columnar Compression stores the values for each column together, and then uses multiple algorithms to compress each column. The database chooses the algorithms based on a variety of factors,

including the data type of the column, the cardinality of the actual values in the column, and the compression level chosen by the user.

As shown in the following graphic, each compression unit can span multiple data blocks. The values for a particular column may or may not span multiple blocks.

**Figure 4-7 Compression Unit**




If Hybrid Columnar Compression does not lead to space savings, then the database stores the data in the `DBMS_COMPRESSION.COMP_BLOCK` format. In this case, the database applies OLTP compression to the blocks, which reside in a Hybrid Columnar Compression segment.

**See Also:**

- "Row Locks (TX)"
- *Oracle Database Licensing Information User Manual* to learn about licensing requirements for Hybrid Columnar Compression
- *Oracle Database Administrator's Guide* to learn how to use Hybrid Columnar Compression
- *Oracle Database SQL Language Reference* for `CREATE TABLE` syntax and semantics
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_COMPRESSION` package

## DML and Hybrid Columnar Compression

Hybrid Columnar Compression has implications for row locking in different types of DML operations.

### Direct Path Loads and Conventional Inserts

When loading data into a table that uses Hybrid Columnar Compression, you can use either conventional inserts or direct path loads. Direct path loads lock the entire table, which reduces concurrency.

Starting with Oracle Database 12c Release 2 (12.2), support is added for conventional array inserts into the Hybrid Columnar Compression format. The advantages of conventional array inserts are:

- Inserted rows use row-level locks, which increases concurrency.
- Automatic Data Optimization (ADO) and Heat Map support Hybrid Columnar Compression for row-level policies. Thus, the database can use Hybrid Columnar Compression for eligible blocks even when DML activity occurs on other parts of the segment.

When the application uses conventional array inserts, Oracle Database stores the rows in compression units when the following conditions are met:

- The table is stored in an ASSM tablespace.
- The compatibility level is 12.2.0.1 or later.
- The table definition satisfies the existing Hybrid Columnar Compression table constraints, including no columns of type `LONG`, and no row dependencies.

Conventional inserts generate redo and undo. Thus, compression units created by conventional DML statement are rolled back or committed along with the DML. The database automatically performs index maintenance, just as for rows that are stored in conventional data blocks.

#### Updates and Deletes

By default, the database locks all rows in the compression unit if an update or delete is applied to any row in the unit. To avoid this issue, you can choose to enable row-level locking for a table. In this case, the database only locks rows that are affected by the update or delete operation.

> **See Also:**
>
> - "Automatic Segment Space Management"
> - "Row Locks (TX)"
> - *Oracle Database Administrator's Guide* to learn how to perform conventional inserts
> - *Oracle Database SQL Language Reference* to learn about the `INSERT` statement

# Overview of Table Clusters

A **table cluster** is a group of tables that share common columns and store related data in the same blocks.

When tables are clustered, a single data block can contain rows from multiple tables. For example, a block can store rows from both the `employees` and `departments` tables rather than from only a single table.

The cluster key is the column or columns that the clustered tables have in common. For example, the `employees` and `departments` tables share the `department_id` column. You specify the cluster key when creating the table cluster and when creating every table added to the table cluster.

The cluster key value is the value of the cluster key columns for a particular set of rows. All data that contains the same cluster key value, such as `department_id=20`, is physically stored together. Each cluster key value is stored only once in the cluster and the cluster index, no matter how many rows of different tables contain the value.

For an analogy, suppose an HR manager has two book cases: one with boxes of employee folders and the other with boxes of department folders. Users often ask for the folders for all employees in a particular department. To make retrieval easier, the manager rearranges all the boxes in a single book case. She divides the boxes by department ID. Thus, all folders for employees in department 20 and the folder for department 20 itself are in one box; the folders for employees in department 100 and the folder for department 100 are in another box, and so on.

Consider clustering tables when they are primarily queried (but not modified) and records from the tables are frequently queried together or joined. Because table clusters store related rows of different tables in the same data blocks, properly used table clusters offer the following benefits over nonclustered tables:

- Disk I/O is reduced for joins of clustered tables.
- Access time improves for joins of clustered tables.
- Less storage is required to store related table and index data because the cluster key value is not stored repeatedly for each row.

Typically, clustering tables is not appropriate in the following situations:

- The tables are frequently updated.
- The tables frequently require a full table scan.
- The tables require truncating.

- Overview of Indexed Clusters
  An **index cluster** is a table cluster that uses an index to locate data. The **cluster index** is a B-tree index on the cluster key. A cluster index must be created before any rows can be inserted into clustered tables.
- Overview of Hash Clusters
  A **hash cluster** is like an indexed cluster, except the index key is replaced with a **hash function**. No separate cluster index exists. In a hash cluster, the data is the index.

# Overview of Indexed Clusters

An **index cluster** is a table cluster that uses an index to locate data. The **cluster index** is a B-tree index on the cluster key. A cluster index must be created before any rows can be inserted into clustered tables.

**Example 4-6 Creating a Table Cluster and Associated Index**

Assume that you create the cluster `employees_departments_cluster` with the cluster key `department_id`, as shown in the following example:

```
CREATE CLUSTER employees_departments_cluster
   (department_id NUMBER(4))
SIZE 512;

CREATE INDEX idx_emp_dept_cluster
   ON CLUSTER employees_departments_cluster;
```

Because the HASHKEYS clause is not specified, employees_departments_cluster is an indexed cluster. The preceding example creates an index named idx_emp_dept_cluster on the cluster key department_id.

**Example 4-7 Creating Tables in an Indexed Cluster**

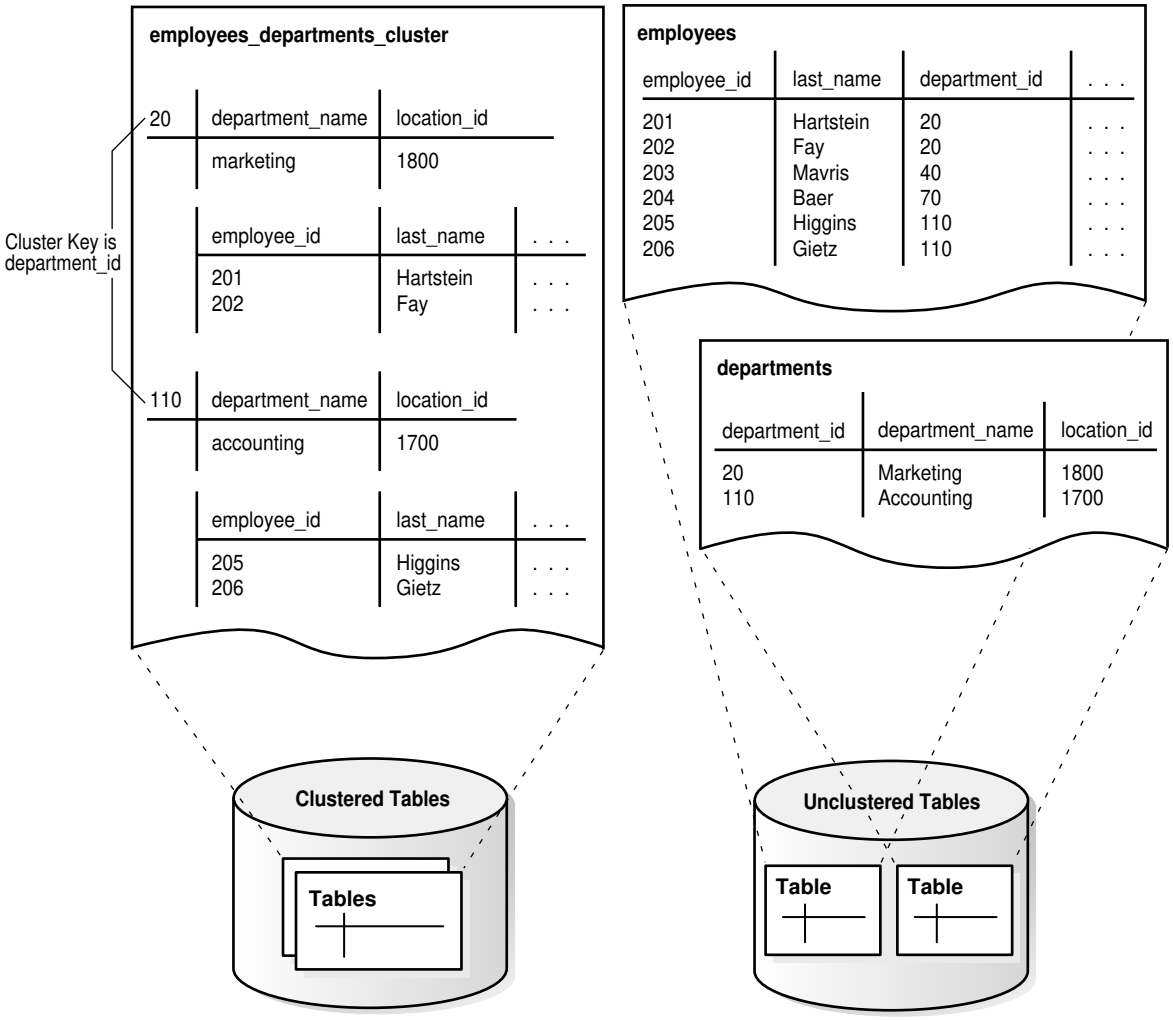
You create the employees and departments tables in the cluster, specifying the department_id column as the cluster key, as follows (the ellipses mark the place where the column specification goes):

```
CREATE TABLE employees ( ... )
   CLUSTER employees_departments_cluster (department_id);

CREATE TABLE departments ( ... )
   CLUSTER employees_departments_cluster (department_id);
```

Assume that you add rows to the employees and departments tables. The database physically stores all rows for each department from the employees and departments tables in the same data blocks. The database stores the rows in a heap and locates them with the index.

Figure 4-8 shows the employees_departments_cluster table cluster, which contains employees and departments. The database stores rows for employees in department 20 together, department 110 together, and so on. If the tables are not clustered, then the database does not ensure that the related rows are stored together.

**Figure 4-8 Clustered Table Data**




The B-tree cluster index associates the cluster key value with the database block address (DBA) of the block containing the data. For example, the index entry for key 20 shows the address of the block that contains data for employees in department 20:

```
20,AADAAAA9d
```

The cluster index is separately managed, just like an index on a nonclustered table, and can exist in a separate tablespace from the table cluster.

> **See Also:**
>
> - "Introduction to Indexes"
> - *Oracle Database Administrator's Guide* to learn how to create and manage indexed clusters
> - *Oracle Database SQL Language Reference* for `CREATE CLUSTER` syntax and semantics

## Overview of Hash Clusters

A **hash cluster** is like an indexed cluster, except the index key is replaced with a **hash function**. No separate cluster index exists. In a hash cluster, the data is the index.

With an indexed table or indexed cluster, Oracle Database locates table rows using key values stored in a separate index. To find or store a row in an indexed table or table cluster, the database must perform at least two I/Os:

- One or more I/Os to find or store the key value in the index
- Another I/O to read or write the row in the table or table cluster

To find or store a row in a hash cluster, Oracle Database applies the hash function to the cluster key value of the row. The resulting hash value corresponds to a data block in the cluster, which the database reads or writes on behalf of the issued statement.

Hashing is an optional way of storing table data to improve the performance of data retrieval. Hash clusters may be beneficial when the following conditions are met:

- A table is queried much more often than modified.
- The hash key column is queried frequently with equality conditions, for example, `WHERE department_id=20`. For such queries, the cluster key value is hashed. The hash key value points directly to the disk area that stores the rows.
- You can reasonably guess the number of hash keys and the size of the data stored with each key value.
- Hash Cluster Creation
  To create a hash cluster, you use the same `CREATE CLUSTER` statement as for an indexed cluster, with the addition of a hash key. The number of hash values for the cluster depends on the hash key.
- Hash Cluster Queries
  In queries of a hash cluster, the database determines how to hash the key values input by the user.
- Hash Cluster Variations
  A single-table hash cluster is an optimized version of a hash cluster that supports only one table at a time. A one-to-one mapping exists between hash keys and rows.
- Hash Cluster Storage
  Oracle Database allocates space for a hash cluster differently from an indexed cluster.

## Hash Cluster Creation

To create a hash cluster, you use the same `CREATE CLUSTER` statement as for an indexed cluster, with the addition of a hash key. The number of hash values for the cluster depends on the hash key.

The cluster key, like the key of an indexed cluster, is a single column or composite key shared by the tables in the cluster. A hash key value is an actual or possible value inserted into the cluster key column. For example, if the cluster key is `department_id`, then hash key values could be 10, 20, 30, and so on.

Oracle Database uses a hash function that accepts an infinite number of hash key values as input and sorts them into a finite number of buckets. Each bucket has a unique numeric ID known as a hash value. Each hash value maps to the database block address for the block that stores the rows corresponding to the hash key value (department 10, 20, 30, and so on).

In the following example, the number of departments that are likely to exist is 100, so `HASHKEYS` is set to `100`:

```
CREATE CLUSTER employees_departments_cluster
   (department_id NUMBER(4))
SIZE 8192 HASHKEYS 100;
```

After you create `employees_departments_cluster`, you can create the `employees` and `departments` tables in the cluster. You can then load data into the hash cluster just as in the indexed cluster.

> **See Also:**
>
> - "Overview of Indexed Clusters"
> - *Oracle Database Administrator's Guide* to learn how to create and manage hash clusters

## Hash Cluster Queries

In queries of a hash cluster, the database determines how to hash the key values input by the user.

For example, users frequently execute queries such as the following, entering different department ID numbers for `p_id`:

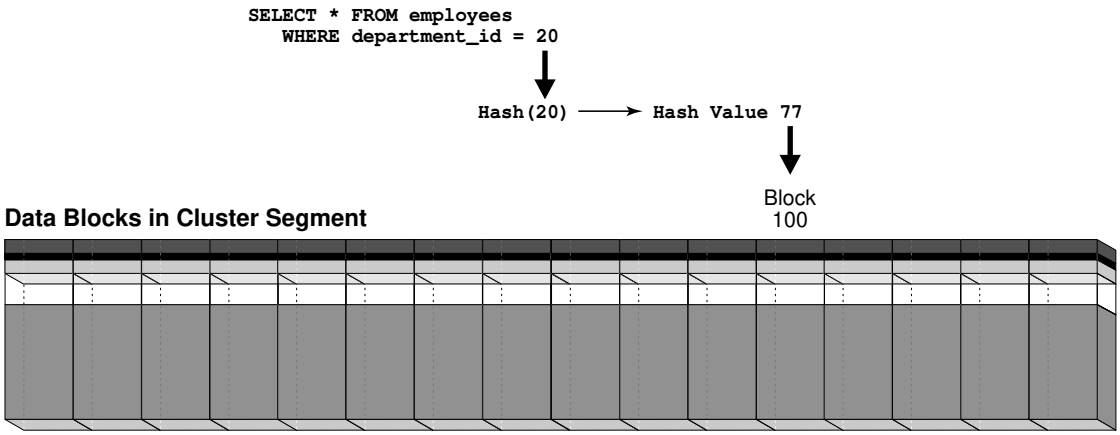
```
SELECT *
FROM   employees
WHERE  department_id = :p_id;

SELECT *
FROM   departments
WHERE  department_id = :p_id;

SELECT *
```

```
FROM   employees e, departments d
WHERE  e.department_id = d.department_id
AND    d.department_id = :p_id;
```

If a user queries employees in `department_id=20`, then the database might hash this value to bucket 77. If a user queries employees in `department_id=10`, then the database might hash this value to bucket 15. The database uses the internally generated hash value to locate the block that contains the employee rows for the requested department.

The following illustration depicts a hash cluster segment as a horizontal row of blocks. As shown in the graphic, a query can retrieve data in a single I/O.

**Figure 4-9 Retrieving Data from a Hash Cluster**




A limitation of hash clusters is the unavailability of range scans on nonindexed cluster keys. Assume no separate index exists for the hash cluster created in Hash Cluster Creation. A query for departments with IDs between 20 and 100 cannot use the hashing algorithm because it cannot hash every possible value between 20 and 100. Because no index exists, the database must perform a full scan.

> **See Also:**
>
> "Index Range Scan"

## Hash Cluster Variations

A single-table hash cluster is an optimized version of a hash cluster that supports only one table at a time. A one-to-one mapping exists between hash keys and rows.

A single-table hash cluster can be beneficial when users require rapid access to a table by primary key. For example, users often look up an employee record in the `employees` table by `employee_id`.

A sorted hash cluster stores the rows corresponding to each value of the hash function in such a way that the database can efficiently return them in sorted order. The database performs the optimized sort internally. For applications that always consume data in sorted order, this

technique can mean faster retrieval of data. For example, an application might always sort on the `order_date` column of the `orders` table.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to create single-table and sorted hash clusters

## Hash Cluster Storage

Oracle Database allocates space for a hash cluster differently from an indexed cluster.

In the example in Hash Cluster Creation, `HASHKEYS` specifies the number of departments likely to exist, whereas `SIZE` specifies the size of the data associated with each department. The database computes a storage space value based on the following formula:

```
HASHKEYS * SIZE / database_block_size
```

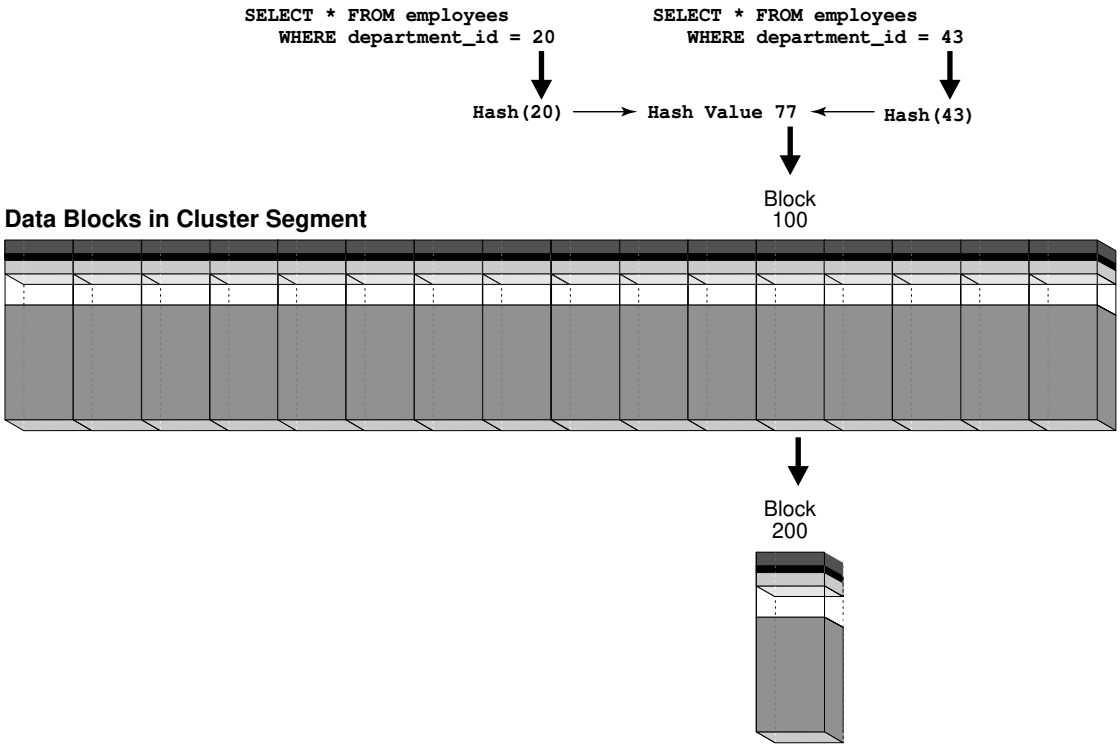
Thus, if the block size is 4096 bytes in the example shown in Hash Cluster Creation, then the database allocates at least 200 blocks to the hash cluster.

Oracle Database does not limit the number of hash key values that you can insert into the cluster. For example, even though `HASHKEYS` is `100`, nothing prevents you from inserting 200 unique departments in the `departments` table. However, the efficiency of the hash cluster retrieval diminishes when the number of hash values exceeds the number of hash keys.

To illustrate the retrieval issues, assume that block 100 in Figure 4-9 is completely full with rows for department 20. A user inserts a new department with `department_id` 43 into the `departments` table. The number of departments exceeds the `HASHKEYS` value, so the database hashes `department_id` 43 to hash value 77, which is the same hash value used for `department_id` 20. Hashing multiple input values to the same output value is called a hash collision.

When users insert rows into the cluster for department 43, the database cannot store these rows in block 100, which is full. The database links block 100 to a new overflow block, say block 200, and stores the inserted rows in the new block. Both block 100 and 200 are now eligible to store data for either department. As shown in Figure 4-10, a query of either department 20 or 43 now requires *two* I/Os to retrieve the data: block 100 and its associated block 200. You can solve this problem by re-creating the cluster with a different `HASHKEYS` value.

**Figure 4-10 Retrieving Data from a Hash Cluster When a Hash Collision Occurs**




> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage space in hash clusters

# Overview of Attribute-Clustered Tables

An **attribute-clustered table** is a heap-organized table that stores data in close proximity on disk based on user-specified clustering directives. The directives specify columns in single or multiple tables.

The directives are as follows:

- The `CLUSTERING ... BY LINEAR ORDER` directive orders data in a table according to specified columns.

  Consider using `BY LINEAR ORDER` clustering, which is the default, when queries qualify the prefix of columns specified in the clustering clause. For example, if queries of `sh.sales` often specify either a customer ID or both customer ID and product ID, then you could cluster data in the table using the linear column order `cust_id`, `prod_id`.

- The `CLUSTERING ... BY INTERLEAVED ORDER` directive orders data in one or more tables using a special algorithm, similar to a Z-order function, that permits multicolumn I/O reduction.

Consider using `BY INTERLEAVED ORDER` clustering when queries specify a variety of column combinations. For example, if queries of `sh.sales` specify different dimensions in different orders, then you can cluster data in the `sales` table according to columns in these dimensions.

Attribute clustering is only available for direct path INSERT operations. It is ignored for conventional DML.

This section contains the following topics:

- Advantages of Attribute-Clustered Tables
- Join Attribute Clustered Tables
- I/O Reduction Using Zones
- Attribute-Clustered Tables with Linear Ordering
- Attribute-Clustered Tables with Interleaved Ordering

- Advantages of Attribute-Clustered Tables
  The primary benefit of attribute-clustered tables is I/O reduction, which can significantly reduce the I/O cost and CPU cost of table scans. I/O reduction occurs either with zones or by reducing physical I/O through closer physical proximity on disk for the clustered values.
- Join Attribute Clustered Tables
  Attribute clustering that is based on joined columns is called **join attribute clustering**. In contrast with table clusters, join attribute clustered tables do not store data from a group of tables in the same database blocks.
- I/O Reduction Using Zones
  A **zone** is a set of contiguous data blocks that stores the minimum and maximum values of relevant columns.
- Attribute-Clustered Tables with Linear Ordering
  A linear ordering scheme for a table divides rows into ranges based on user-specified attributes in a specific order. Oracle Database supports linear ordering on single or multiple tables that are connected through a primary-foreign key relationship.
- Attribute-Clustered Tables with Interleaved Ordering
  Interleaved ordering uses a technique that is similar to a *Z-order*.

## Advantages of Attribute-Clustered Tables

The primary benefit of attribute-clustered tables is I/O reduction, which can significantly reduce the I/O cost and CPU cost of table scans. I/O reduction occurs either with zones or by reducing physical I/O through closer physical proximity on disk for the clustered values.

An attribute-clustered table has the following advantages:

- You can cluster fact tables based on dimension columns in star schemas.

  In star schemas, most queries qualify dimension tables and not fact tables, so clustering by fact table columns is not effective. Oracle Database supports clustering on columns in dimension tables.
- I/O reduction can occur in several different scenarios:
  - When used with Oracle Exadata Storage Indexes, Oracle In-Memory min/max pruning, or zone maps
  - In OLTP applications for queries that qualify a prefix and use attribute clustering with linear order

  - On a subset of the clustering columns for `BY INTERLEAVED ORDER` clustering

- Attribute clustering can improve data compression, and in this way indirectly improve table scan costs.

  When the same values are close to each other on disk, the database can more easily compress them.

- Oracle Database does not incur the storage and maintenance cost of an index.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more advantages of attribute-clustered tables

## Join Attribute Clustered Tables

Attribute clustering that is based on joined columns is called **join attribute clustering**. In contrast with table clusters, join attribute clustered tables do not store data from a group of tables in the same database blocks.

For example, consider an attribute-clustered table, `sales`, joined with a dimension table, `products`. The `sales` table contains only rows from the `sales` table, but the ordering of the rows is based on the values of columns joined from `products` table. The appropriate join is executed during data movement, direct path insert, and `CREATE TABLE AS SELECT` operations. In contrast, if `sales` and `products` were in a standard table cluster, the data blocks would contain rows from both tables.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* to learn more about join attribute clustering

## I/O Reduction Using Zones

A **zone** is a set of contiguous data blocks that stores the minimum and maximum values of relevant columns.

When a SQL statement contains predicates on columns stored in a zone, the database compares the predicate values to the minimum and maximum stored in the zone. In this way, the database determines which zones to read during SQL execution.

I/O reduction is the ability to skip table or index blocks that do not contain data that the database needs to satisfy a query. This reduction can significantly reduce the I/O and CPU cost of table scans.

- Purpose of Zones
  For a loose analogy of zones, consider a sales manager who uses a bookcase of pigeonholes, which are analogous to data blocks.

- Zone Maps
  A **zone map** is an independent access structure that divides data blocks into zones. Oracle Database implements each zone map as a type of **materialized view**.
- Zone Map Creation
  Basic zone maps are created either manually or automatically.
- How a Zone Map Works: Example
  This example illustrates how a zone map can prune data in a query whose predicate contains a constant.

## Purpose of Zones

For a loose analogy of zones, consider a sales manager who uses a bookcase of pigeonholes, which are analogous to data blocks.

Each pigeonhole has receipts (rows) describing shirts sold to a customer, ordered by ship date. In this analogy, a zone map is like a stack of index cards. Each card corresponds to a "zone" (contiguous range) of pigeonholes, such as pigeonholes 1-10. For each zone, the card lists the minimum and maximum ship dates for the receipts stored in the zone.

When someone wants to know which shirts shipped on a certain date, the manager flips the cards until she comes to the date range that contains the requested date, notes the pigeonhole zone, and then searches only pigeonholes in this zone for the requested receipts. In this way, the manager avoids searching every pigeonhole in the bookcase for the receipts.

## Zone Maps

A **zone map** is an independent access structure that divides data blocks into zones. Oracle Database implements each zone map as a type of **materialized view**.

Like indexes, zone maps can reduce the I/O and CPU costs of table scans. When a SQL statement contains predicates on columns in a zone map, the database compares the predicate values to the minimum and maximum table column values stored in each zone to determine which zones to read during SQL execution.

A **basic zone map** is defined on a single table and maintains the minimum and maximum values of some columns of this table. A **join zone map** is defined on a table that has an outer join to one or more other tables and maintains the minimum and maximum values of some columns in the other tables. Oracle Database maintains both types of zone map automatically.

At most one zone map can exist on a table. In the case of a partitioned table, one zone map exists for all partitions and subpartitions. A zone map of a partitioned table also keeps track of the minimum and maximum values per zone, per partition, and per subpartition. Zone map definitions can include minimum and maximum values of dimension columns provided the table has an outer join with the dimension tables.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for an overview of zone maps

## Zone Map Creation

Basic zone maps are created either manually or automatically.

- Manual Zone Maps
  You can create, drop, and maintain zone maps using DDL statements.
- Automatic Zone Maps
  Oracle Database can create basic zone maps automatically. These are known as automatic zone maps.

## Manual Zone Maps

You can create, drop, and maintain zone maps using DDL statements.

Whenever you specify the `CLUSTERING` clause in a `CREATE TABLE` or `ALTER TABLE` statement, the database automatically creates a zone map on the specified clustering columns. The zone map correlates minimum and maximum values of columns with consecutive data blocks in the attribute-clustered table. Attribute-clustered tables use zone maps to perform I/O reduction.

You can also create zone maps explicitly by using the `CREATE MATERIALIZED ZONEMAP` statement. In this case, you can create zone maps for use with or without attribute clustering. For example, you can create a zone map on a table whose rows are naturally ordered on a set of columns, such as a stock trade table whose trades are ordered by time.

**See Also:**

- "Overview of Materialized Views"
- *Oracle Database Data Warehousing Guide* to learn more how to create zone maps

## Automatic Zone Maps

Oracle Database can create basic zone maps automatically. These are known as automatic zone maps.

Oracle Database can create basic zone maps automatically for both partitioned and nonpartitioned tables. A background process automatically maintains zone maps created in this way.

Use the `DBMS_AUTO_ZONEMAP` procedure to enable automatic zone maps:

```
EXEC DBMS_AUTO_ZONEMAP.CONFIGURE('AUTO_ZONEMAP_MODE','ON')
```

**See Also:**

- *Oracle Database Data Warehousing Guide* to learn more about managing automatic zone maps using the `DBMS_AUTO_ZONEMAP` package
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_AUTO_ZONEMAP` package
- *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

## How a Zone Map Works: Example

This example illustrates how a zone map can prune data in a query whose predicate contains a constant.

Assume you create the following `lineitem` table:

```
CREATE TABLE lineitem
  ( orderkey      NUMBER        ,
    shipdate      DATE          ,
    receiptdate   DATE          ,
    destination   VARCHAR2(50)  ,
    quantity      NUMBER        );
```

The table `lineitem` contains 4 data blocks with 2 rows per block. Table 4-4 shows the 8 rows of the table.

**Table 4-4 Data Blocks for lineitem Table**

| Block | orderkey | shipdate | receiptdate | destination | quantity |
|---|---|---|---|---|---|
| 1 | 1 | 1-1-2014 | 1-10-2014 | San_Fran | 100 |
| 1 | 2 | 1-2-2014 | 1-10-2014 | San_Fran | 200 |
| 2 | 3 | 1-3-2014 | 1-9-2014 | San_Fran | 100 |
| 2 | 4 | 1-5-2014 | 1-10-2014 | San_Diego | 100 |
| 3 | 5 | 1-10-2014 | 1-15-2014 | San_Fran | 100 |
| 3 | 6 | 1-12-2014 | 1-16-2014 | San_Fran | 200 |
| 4 | 7 | 1-13-2014 | 1-20-2014 | San_Fran | 100 |
| 4 | 8 | 1-15-2014 | 1-30-2014 | San_Jose | 100 |

You can use the `CREATE MATERIALIZED ZONEMAP` statement to create a zone map on the `lineitem` table. Each zone contains 2 blocks and stores the minimum and maximum of the `orderkey`, `shipdate`, and `receiptdate` columns. Table 4-5 shows the zone map.

**Table 4-5 Zone Map for lineitem Table**

| Block Range | min orderkey | max orderkey | min shipdate | max shipdate | min receiptdate | max receiptdate |
|---|---|---|---|---|---|---|
| 1-2 | 1 | 4 | 1-1-2014 | 1-5-2014 | 1-9-2014 | 1-10-2014 |
| 3-4 | 5 | 8 | 1-10-2014 | 1-15-2014 | 1-15-2014 | 1-30-2014 |

When you execute the following query, the database can read the zone map and then scan only blocks 1 and 2, and therefore skip blocks 3 and 4, because the date `1-3-2014` falls between the minimum and maximum dates:

```
SELECT * FROM lineitem WHERE shipdate = '1-3-2014';
```

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* to learn how to use zone maps
> - *Oracle Database SQL Language Reference* for syntax and semantics of the `CREATE MATERIALIZED ZONEMAP` statement

## Attribute-Clustered Tables with Linear Ordering

A linear ordering scheme for a table divides rows into ranges based on user-specified attributes in a specific order. Oracle Database supports linear ordering on single or multiple tables that are connected through a primary-foreign key relationship.

For example, the `sales` table divides the `cust_id` and `prod_id` columns into ranges, and then clusters these ranges together on disk. When you specify the `BY LINEAR ORDER` directive for a table, significant I/O reduction can occur when a predicate specifies either the prefix column or all columns in the directive.

Assume that queries of `sales` often specify either a customer ID or a combination of a customer ID and product ID. You can create an attribute-clustered table so that such queries benefit from I/O reduction:

```
CREATE TABLE sales
(
   prod_id     NOT NULL NUMBER
,  cust_id     NOT NULL NUMBER
,  amount_sold NUMBER(10,2) ...
)
CLUSTERING
  BY LINEAR ORDER (cust_id, prod_id)
  YES ON LOAD YES ON DATA MOVEMENT
  WITH MATERIALIZED ZONEMAP;
```

Queries that qualify both columns `cust_id` and `prod_id`, or the prefix `cust_id` experience I/O reduction. Queries that qualify `prod_id` only do not experience significant I/O reduction because `prod_id` is the suffix of the `BY LINEAR ORDER` clause. The following examples show how the database can reduce I/O during table scans.

**Example 4-8 Specifying Only cust_id**

An application issues the following query:

```
SELECT * FROM sales WHERE cust_id = 100;
```

Because the `sales` table is a `BY LINEAR ORDER` cluster, the database must only read the zones that include the `cust_id` value of `100`.

**Example 4-9 Specifying prod_id and cust_id**

An application issues the following query:

```
SELECT * FROM sales WHERE cust_id = 100 AND prod_id = 2300;
```

Because the `sales` table is a `BY LINEAR ORDER` cluster, the database must only read the zones that include the `cust_id` value of `100` and `prod_id` value of `2300`.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* to learn how to cluster tables using linear ordering
> - *Oracle Database SQL Language Reference* for syntax and semantics of the `BY LINEAR ORDER` clause

## Attribute-Clustered Tables with Interleaved Ordering

Interleaved ordering uses a technique that is similar to a *Z-order*.

Interleaved ordering enables the database to prune I/O based on any subset of predicates in the clustering columns. Interleaved ordering is useful for dimensional hierarchies in a data warehouse.

As with attribute-clustered tables with linear ordering, Oracle Database supports interleaved ordering on single or multiple tables that are connected through a primary-foreign key relationship. Columns in tables other than the attribute-clustered table must be linked by foreign key and joined to the attribute-clustered table.

Large data warehouses frequently organize data in a star schema. A dimension table uses a parent-child hierarchy and is connected to a fact table by a foreign key. Clustering a fact table by interleaved order enables the database to use a special function to skip values in dimension columns during table scans.

**Example 4-10 Interleaved Ordering Example**

Suppose your data warehouse contains a `sales` fact table and its two dimension tables: `customers` and `products`. Most queries have predicates on the `customers` table hierarchy `(cust_state_province, cust_city)` and the products hierarchy `(prod_category, prod_subcategory)`. You can use interleaved ordering for the `sales` table as shown in the partial statement in the following example:

```
CREATE TABLE sales
(
   prod_id NUMBER NOT NULL
,  cust_id NUMBER NOT NULL
,  amount_sold NUMBER(10,2) ...
)
CLUSTERING sales
   JOIN products ON (sales.prod_id = products.prod_id)
   JOIN customers ON (sales.cust_id = customers.cust_id)
   BY INTERLEAVED ORDER
   (
     (  products.prod_category
     ,  products.prod_subcategory
     ),
     (  customers.cust_state_province
     ,  customers.cust_city
```

```
    )
  )
WITH MATERIALIZED ZONEMAP;
```

> **Note:**
>
> The columns specified in the `BY INTERLEAVED ORDER` clause need not be in actual dimension tables, but they must be connected through a primary-foreign key relationship.

Suppose an application queries the `sales`, `products`, and `customers` tables in a join. The query specifies the `customers.prod_category` and `customers_cust_state_province` columns in the predicate as follows:

```
SELECT cust_city, prod_sub_category, SUM(amount_sold)
FROM   sales, products, customers
WHERE  sales.prod_id = products.prod_id
AND    sales.cust_id = customers.cust_id
AND    customers.prod_category = 'Boys'
AND    customers.cust_state_province = 'England - Norfolk'
GROUP BY cust_city, prod_sub_category;
```

In the preceding query, the `prod_category` and `cust_state_province` columns are part of the clustering definition shown in the `CREATE TABLE` example. During the scan of the `sales` table, the database can consult the zone map and access only the rowids in this zone.

> **See Also:**
>
> - "Overview of Dimensions"
> - *Oracle Database Data Warehousing Guide* to learn how to cluster tables using interleaved ordering
> - *Oracle Database SQL Language Reference* for syntax and semantics of the `BY INTERLEAVED ORDER` clause

# Overview of Temporary Tables

A **temporary table** holds data that exists only for the duration of a transaction or session.

Data in a temporary table is private to the session. Each session can only see and modify its own data.

You can create either a **global temporary table** or a **private temporary table**. The following table shows the essential differences between them.

**Table 4-6 Temporary Table Characteristics**

| Characteristic | Global | Private |
|---|---|---|
| Naming rules | Same as for permanent tables | Must be prefixed with `ORA$PTT_` |
| Visibility of table definition | All sessions | Only the session that created the table |
| Storage of table definition | Disk | Memory only |
| Types | Transaction-specific (`ON COMMIT DELETE ROWS`) or session-specific (`ON COMMIT PRESERVE ROWS`) | Transaction-specific (`ON COMMIT DROP DEFINITION`) or session-specific (`ON COMMIT PRESERVE DEFINITION`) |

A third type of temporary table, known as a **cursor-duration temporary table**, is created by the database automatically for certain types of queries.

- Purpose of Temporary Tables
  Temporary tables are useful in applications where a result set must be buffered.
- Segment Allocation in Temporary Tables
  Like permanent tables, global temporary tables are persistent objects that are statically defined in the data dictionary. For private temporary tables, metadata exists only in memory, but can reside in the temporary tablespace on disk.
- Temporary Table Creation
  The `CREATE ... TEMPORARY TABLE` statement creates a temporary table.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* to learn more about cursor-duration temporary tables

## Purpose of Temporary Tables

Temporary tables are useful in applications where a result set must be buffered.

For example, a scheduling application enables college students to create optional semester course schedules. A row in a global temporary table represents each schedule. During the session, the schedule data is private. When the student chooses a schedule, the application moves the row for the chosen schedule to a permanent table. At the end of the session, the database automatically drops the schedule data that was in the global temporary table.

Private temporary tables are useful for dynamic reporting applications. For example, a customer resource management (CRM) application might connect as the same user indefinitely, with multiple sessions active at the same time. Each session creates a private temporary table named `ORA$PTT_crm` for each new transaction. The application can use the same table name for every session, but change the definition. The data and definition are visible only to the session. The table definition persists until the transaction ends or the table is manually dropped.

## Segment Allocation in Temporary Tables

Like permanent tables, global temporary tables are persistent objects that are statically defined in the data dictionary. For private temporary tables, metadata exists only in memory, but can reside in the temporary tablespace on disk.

For global and private temporary tables, the database allocates temporary segments when a session first inserts data. Until data is loaded in a session, the table appears empty. For transaction-specific temporary tables, the database deallocates temporary segments at the end of the transaction. For session-specific temporary tables, the database deallocates temporary segments at the end of the session.

> **See Also:**
>
> "Temporary Segments"

## Temporary Table Creation

The `CREATE ... TEMPORARY TABLE` statement creates a temporary table.

Specify either `GLOBAL TEMPORARY TABLE` or `PRIVATE TEMPORARY TABLE`. In both cases, the `ON COMMIT` clause specifies whether the table data is transaction-specific (default) or session-specific. You create a temporary table for the database itself, not for every PL/SQL stored procedure.

You can create indexes for global (not private) temporary tables with the `CREATE INDEX` statement. These indexes are also temporary. The data in the index has the same session or transaction scope as the data in the temporary table. You can also create a view or trigger on a global temporary table.

> **See Also:**
>
> - "Overview of Views"
> - "Overview of Triggers"
> - *Oracle Database Administrator's Guide* to learn how to create and manage temporary tables
> - *Oracle Database SQL Language Reference* for `CREATE ... TEMPORARY TABLE` syntax and semantics

# Overview of External Tables

An **external table** accesses data in external sources as if this data were in a table in the database.

The data can be in any format for which an access driver is provided. You can use SQL (serial or parallel), PL/SQL, and Java to query external tables.

- Purpose of External Tables
  External tables are useful when an Oracle database application must access non-relational data.
- Data in Object Stores
  External tables can be used to access data in object stores.
- External Table Access Drivers
  An **access driver** is an API that interprets the external data for the database. The access driver runs inside the database, which uses the driver to read the data in the external table. The access driver and the external table layer are responsible for performing the transformations required on the data in the data file so that it matches the external table definition.
- External Table Creation
  Internally, creating an external table means creating metadata in the data dictionary. Unlike an ordinary table, an external table does not describe data stored in the database, nor does it describe how data is stored externally. Rather, external table metadata describes how the external table layer must *present* data to the database.

# Purpose of External Tables

External tables are useful when an Oracle database application must access non-relational data.

For example, a SQL-based application may need to access a text file whose records are in the following form:

```
100,Steven,King,SKING,515.123.4567,17-JUN-03,AD_PRES,31944,150,90
101,Neena,Kochhar,NKOCHHAR,515.123.4568,21-SEP-05,AD_VP,17000,100,90
102,Lex,De Haan,LDEHAAN,515.123.4569,13-JAN-01,AD_VP,17000,100,90
```

You could create an external table, copy the text file to the location specified in the external table definition, and then use SQL to query the records in the text file. Similarly, you could use external tables to give read-only access to JSON documents or LOBs.

In data warehouse environments, external tables are valuable for performing extraction, transformation, and loading (ETL) tasks. For example, external tables enable you to pipeline the data loading phase with the transformation phase. This technique eliminates the need to stage data inside the database in preparation for further processing inside the database.

You can partition external tables on virtual or non-virtual columns. Also, you can create a hybrid partitioned table, where some partitions are internal and some external. Like internal partitions, external benefit from performance enhancements such as partition pruning and partition-wise joins. For example, you could use partitioned external tables to analyze large volumes of non-relational data stored on Hadoop Distributed File System (HDFS) or a NoSQL database.

> **See Also:**
>
> "Partitioned Tables"

## Data in Object Stores

External tables can be used to access data in object stores.

In addition to supporting access to external data residing in operating system files and Big Data sources, Oracle supports access to external data in object stores. Object storage is common in the Cloud and provides a flat architecture to manage individual objects, any type of unstructured data with metadata, by grouping them in simple containers. Although object storage is predominantly a data storage architecture in the Cloud, it is also available as on-premises storage hardware.

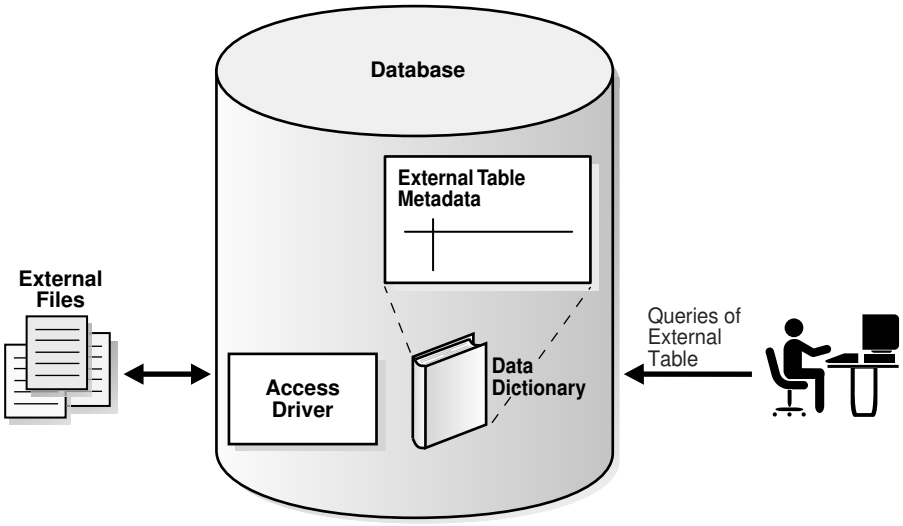
You can access data in object stores by using the `DBMS_CLOUD` package or by manually defining external tables. Oracle strongly recommends using the `DBMS_CLOUD` package because it provides additional functionality and is fully compatible with Oracle Autonomous Database.

## External Table Access Drivers

An **access driver** is an API that interprets the external data for the database. The access driver runs inside the database, which uses the driver to read the data in the external table. The access driver and the external table layer are responsible for performing the transformations required on the data in the data file so that it matches the external table definition.

The following figure represents SQL access of external data.

**Figure 4-11 External Tables**




Oracle provides the following access drivers for external tables:

- `ORACLE_LOADER` (default)

  Enables access to external files using most of the formats supported by SQL*Loader. You cannot create, update, or append to an external file using the `ORACLE_LOADER` driver.

- `ORACLE_DATAPUMP`

  Enables you to unload or load external data. An unload operation reads data from the database and inserts the data into an external table, represented by one or more external

files. After external files are created, the database cannot update or append data to them. A load operation reads an external table and loads its data into a database.

- `ORACLE_HDFS`

  Enables the extraction of data stored in a Hadoop Distributed File System (HDFS).

- `ORACLE_HIVE`

  Enables access to data stored in an Apache Hive database. The source data can be stored in HDFS, HBase, Cassandra, or other systems. Unlike the other access drivers, you cannot specify a location because `ORACLE_HIVE` obtains location information from an external metadata store.

- `ORACLE_BIGDATA`

  Enables read-only access to data stored in both structured and unstructured formats, including Apache Parquet, Apache Avro, Apache ORC, and text formats. You can also use this driver to query local data, which is useful for testing and smaller data sets.

## External Table Creation

Internally, creating an external table means creating metadata in the data dictionary. Unlike an ordinary table, an external table does not describe data stored in the database, nor does it describe how data is stored externally. Rather, external table metadata describes how the external table layer must *present* data to the database.

A `CREATE TABLE ... ORGANIZATION EXTERNAL` statement has two parts. The external table definition describes the column types. This definition is like a view that enables SQL to query external data without loading it into the database. The second part of the statement maps the external data to the columns.

External tables are read-only unless created with `CREATE TABLE AS SELECT` with the `ORACLE_DATAPUMP` access driver. Restrictions for external tables include no support for indexed columns and column objects.

> **See Also:**
>
> - *Oracle Database Utilities* to learn about external tables
> - *Oracle Database Administrator's Guide* to learn about managing external tables, external connections, and directory objects
> - *Oracle Database SQL Language Reference* for information about creating and querying external tables

# Overview of Blockchain Tables

A **blockchain table** is an append-only table designed for centralized blockchain applications.

In Oracle Blockchain Table, peers are database users who trust the database to maintain a tamper-resistant ledger. The ledger is implemented as a blockchain table, which is defined and managed by the application. Existing applications can protect against fraud without requiring a new infrastructure or programming model. Although transaction throughput is lower than for a standard table, performance for a blockchain table is better than for a decentralized blockchain.

A blockchain table is append-only because the only permitted DML are `INSERT` commands. The table disallows `UPDATE`, `DELETE`, `MERGE`, `TRUNCATE`, and direct-path loads. Database transactions can span blockchain tables and standard tables. For example, a single transaction can insert rows into a standard table and two different blockchain tables.

- Row Chains
  In a blockchain table, a **row chain** is a series of rows linked together with a hashing scheme.
- Row Content
  The **row content** is a contiguous sequence of bytes containing the column data of the row and the hash value of the previous row in the chain.
- User Interface for Blockchain Tables
  Like a standard table, a blockchain table is created by SQL and supports scalar data types, LOBs, and partitions. You can also create indexes and triggers for blockchain tables.

## Row Chains

In a blockchain table, a **row chain** is a series of rows linked together with a hashing scheme.

A row chain is identified by unique combination of database instance ID and chain ID. A row in a blockchain table belongs to exactly one row chain. A single table supports multiple row chains.

> **Note:**
>
> A chained row in a standard table is orthogonal to a row chain in a blockchain table. Only the word "chain" is the same.

Every row in a chain has a unique sequence number. The database sequences the rows using an SHA2-512 hash computation on the rows of each chain. The hash for every inserted row is derived from the hash value of the previously inserted row in the chain and the **row content** of the inserted row.

## Row Content

The **row content** is a contiguous sequence of bytes containing the column data of the row and the hash value of the previous row in the chain.

When you create a blockchain table, the database creates several hidden columns. For example, you might create the blockchain table `bank_ledger` with the columns `bank` and `deposit`:

```
CREATE BLOCKCHAIN TABLE bank_ledger (bank VARCHAR2 (128), deposit NUMBER)
  NO DROP UNTIL 31 DAYS IDLE
  NO DELETE UNTIL 31 DAYS AFTER INSERT
  HASHING USING "SHA2_512" VERSION "v1";
```

The database automatically creates hidden columns with the prefix `ORABCTAB`: `ORABCTAB_INST_ID$`, `ORABCTAB_CHAIN_ID$`, `ORABCTAB_SEQ_NUM$`, and others. These hidden columns, which you cannot alter or manage, implement the anti-tampering algorithm. This

algorithm avoids deadlocks by acquiring unique, table-level locks in a specific order at commit time.

> **Note:**
>
> Row content for blockchain tables is stored in standard data blocks. In Oracle Database 21c, blockchain tables do not support table clusters.

The instance ID, chain ID, and sequence number uniquely identify a row. Each row has a platform-independent SHA2-512 hash that is stored in hidden column `ORABCTAB_HASH$`. The hash is based on the content of the inserted row and the hash of the previous row in the chain.

The data format for the column value of a row consists of bytes from the column metadata and content. The column metadata is a 20-byte structure that describes characteristics such as position in the table, data type, null status, and byte length. The column content is the set of bytes representing the value in a row. For example, the ASCII representation of the value `Chase` is `43 68 61 73 65`. You can use the `DUMP` function in SQL to obtain both column metadata and content.

The row content for a hash computation includes the column data formats from multiple columns: the hash value in the previous row in the chain, the user-defined columns, and a fixed number of hidden columns.

## User Interface for Blockchain Tables

Like a standard table, a blockchain table is created by SQL and supports scalar data types, LOBs, and partitions. You can also create indexes and triggers for blockchain tables.

To create a blockchain table, use a `CREATE BLOCKCHAIN TABLE` statement. A blockchain table has a retention period specified by the `NO DROP UNTIL` *n* `DAYS IDLE` clause. You can remove the table by using `DROP TABLE`.

Oracle Blockchain Table supports the following interfaces:

- The `DBMS_BLOCKCHAIN_TABLE` package enables you to perform various operations on table rows. For example, to apply a signature to the content of a previously inserted row, use the `SIGN_ROW` procedure. To verify that the rows have not been tampered with, use `VERIFY_ROWS`. To remove rows after the retention period (specified by the `NO DELETE` clause) has passed, use `DELETE_ROWS`.
- The `DBMS_TABLE_DATA` package provides procedures to retrieve the byte value of a column. You can retrieve the row content for row data on which the hash or user signature is computed.
- The `DBA_BLOCKCHAIN_TABLES` view shows table metadata such as the row retention period, inactivity period before a table drop is permitted, and hash algorithm.

**Note:**

- *Oracle Database Administrator's Guide* to learn how to manage blockchain tables
- *Oracle Database PL/SQL Packages and Types Reference*to learn about the `DBMS_BLOCKCHAIN_TABLE` package
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_TABLE_DATA` package
- *Oracle Database Reference* to learn about the `DBA_BLOCKCHAIN_TABLES` view

# Overview of Immutable Tables

Immutable tables are read-only tables that prevent unauthorized data modifications by insiders and accidental data modifications resulting from human errors.

Unauthorized modifications can be attempted by compromised or rogue employees who have access to insider credentials.

New rows can be added to an immutable table, but existing rows cannot be modified. You must specify a retention period both for the immutable table and for rows within the immutable table. Rows become obsolete after the specified row retention period. Only obsolete rows can be deleted from the immutable table.

Immutable tables contain system-generated hidden columns. The columns are the same as those for blockchain tables. When a row is inserted, a non-NULL value is set for the `ORABCTAB_CREATION_TIME$` and `ORABCTAB_USER_NUMBER$` columns. The value of remaining system-generated hidden columns is set to `NULL`.

Using immutable tables requires no changes to existing applications.

# Overview of Object Tables

An **object table** is a special kind of table in which each row represents an object.

An Oracle **object type** is a user-defined type with a name, attributes, and methods. Object types make it possible to model real-world entities such as customers and purchase orders as objects in the database.

An object type defines a logical structure, but does not create storage. The following example creates an object type named `department_typ`:

```
CREATE TYPE department_typ AS OBJECT
   ( d_name     VARCHAR2(100),
     d_address  VARCHAR2(200) );
/
```

The following example creates an object table named `departments_obj_t` of the object type `department_typ`, and then inserts a row into the table. The attributes (columns) of the `departments_obj_t` table are derived from the definition of the object type.

```
CREATE TABLE departments_obj_t OF department_typ;
INSERT INTO departments_obj_t VALUES ('hr', '10 Main St, Sometown, CA');
```

Like a relational column, an object table can contain rows of just one kind of thing, namely, object instances of the same declared type as the table. By default, every row object in an object table has an associated logical object identifier (OID) that uniquely identifies it in an object table. The OID column of an object table is a hidden column.

**See Also:**

- *Oracle Database Object-Relational Developer's Guide* to learn about object-relational features in Oracle Database
- *Oracle Database SQL Language Reference* for `CREATE TYPE` syntax and semantics

5

# Indexes and Index-Organized Tables

Indexes are schema objects that can speed access to table rows. Index-organized tables are tables stored in an index structure.

This chapter contains the following sections:


- Introduction to Indexes
- Overview of B-Tree Indexes
- Overview of Bitmap Indexes
- Overview of Function-Based Indexes
- Overview of Application Domain Indexes
- Overview of Index-Organized Tables


- Introduction to Indexes
  An **index** is an optional structure, associated with a table or **table cluster**, that can sometimes speed data access.
- Overview of B-Tree Indexes
  B-trees, short for *balanced trees*, are the most common type of database index. A **B-tree index** is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- Overview of Bitmap Indexes
  In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows.
- Overview of Function-Based Indexes
  A **function-based index** computes the value of a function or expression involving one or more columns and stores it in an index. A function-based index can be either a B-tree or a bitmap index.
- Overview of Application Domain Indexes
  An **application domain index** is a customized index specific to an application.
- Overview of Index-Organized Tables
  An **index-organized table** is a table stored in a variation of a B-tree index structure. In contrast, a **heap-organized table** inserts rows where they fit.

## Introduction to Indexes

An **index** is an optional structure, associated with a table or **table cluster**, that can sometimes speed data access.

Indexes are schema objects that are logically and physically independent of the data in the objects with which they are associated. Thus, you can drop or create an index without physically affecting the indexed table.

> **Note:**
>
> If you drop an index, then applications still work. However, access of previously indexed data can be slower.

For an analogy, suppose an HR manager has a shelf of cardboard boxes. Folders containing employee information are inserted randomly in the boxes. The folder for employee Whalen (ID 200) is 10 folders up from the bottom of box 1, whereas the folder for King (ID 100) is at the bottom of box 3. To locate a folder, the manager looks at every folder in box 1 from bottom to top, and then moves from box to box until the folder is found. To speed access, the manager could create an index that sequentially lists every employee ID with its folder location:

```
ID 100: Box 3, position 1 (bottom)
ID 101: Box 7, position 8
ID 200: Box 1, position 10
.
.
.
```

Similarly, the manager could create separate indexes for employee last names, department IDs, and so on.

This section contains the following topics:

- Advantages and Disadvantages of Indexes
- Index Usability and Visibility
- Keys and Columns
- Composite Indexes
- Unique and Nonunique Indexes
- Types of Indexes
- How the Database Maintains Indexes
- Index Storage

- Advantages and Disadvantages of Indexes
  The absence or presence of an index does not require a change in the wording of any SQL statement.
- Index Usability and Visibility
  Indexes are usable (default) or unusable, visible (default) or invisible.
- Keys and Columns
  A **key** is a set of columns or expressions on which you can build an index.
- Composite Indexes
  A **composite index**, also called a **concatenated index**, is an index on multiple columns in a table.
- Unique and Nonunique Indexes
  Indexes can be unique or nonunique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns.

- Types of Indexes
  Oracle Database provides several indexing schemes, which provide complementary performance functionality.
- How the Database Maintains Indexes
  The database automatically maintains and uses indexes after they are created.
- Index Storage
  Oracle Database stores index data in an index segment.

## Advantages and Disadvantages of Indexes

The absence or presence of an index does not require a change in the wording of any SQL statement.

An index is a fast access path to a single row of data. It affects only the speed of execution. Given a data value that has been indexed, the index points directly to the location of the rows containing that value.

When an index exists on one or more columns of a table, the database can in some cases retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of reducing disk I/O. If a heap-organized table has no indexes, then the database must perform a full table scan to find a value. For example, a query of location `2700` in the unindexed `hr.departments` table requires the database to search every row in every block. This approach does not scale well as data volumes increase.

Disadvantages of indexes include the following:

- Creating indexes manually often requires deep knowledge of the data model, application, and data distribution.
- As the data changes, you must revisit previous decisions about indexes. An index might stop being useful, or new indexes might be required.
- Indexes occupy disk space.
- The database must update the index when DML occurs on the indexed data, which creates performance overhead.

> **Note:**
>
> Starting in Oracle Database 19c, Oracle Database can constantly monitor the application workload, creating and managing indexes automatically. Automated indexing is implemented as a database task that runs at a fixed interval.

Consider creating an index in the following situations:

- The indexed columns are queried frequently and return a small percentage of the total number of rows in the table.
- A referential integrity constraint exists on the indexed column or columns. The index is a means to avoid a full table lock that would otherwise be required if you update the parent table primary key, merge into the parent table, or delete from the parent table.
- A unique key constraint will be placed on the table and you want to manually specify the index and all index options.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn more about automated indexing
> - *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

## Index Usability and Visibility

Indexes are usable (default) or unusable, visible (default) or invisible.

These properties are defined as follows:

- Usability

  An unusable index, which is ignored by the optimizer, is not maintained by DML operations. An unusable index can improve the performance of bulk loads. Instead of dropping an index and later re-creating it, you can make the index unusable and then rebuild it. Unusable indexes and index partitions do not consume space. When you make a usable index unusable, the database drops its index segment.

- Visibility

  An invisible index is maintained by DML operations, but is not used by default by the optimizer. Making an index invisible is an alternative to making it unusable or dropping it. Invisible indexes are especially useful for testing the removal of an index before dropping it or using indexes temporarily without affecting the overall application.

> **See Also:**
>
> "Overview of the Optimizer" to learn about how the optimizer select execution plans

## Keys and Columns

A **key** is a set of columns or expressions on which you can build an index.

Although the terms are often used interchangeably, indexes and keys are different. Indexes are structures stored in the database that users manage using SQL statements. Keys are strictly a logical concept.

The following statement creates an index on the `customer_id` column of the sample table `oe.orders`:

```
CREATE INDEX ord_customer_ix ON orders (customer_id);
```

In the preceding statement, the `customer_id` column is the index key. The index itself is named `ord_customer_ix`.

> **Note:**
>
> Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key.

> **See Also:**
>
> - "Data Integrity"
> - *Oracle Database SQL Language Reference* `CREATE INDEX` syntax and semantics

## Composite Indexes

A **composite index**, also called a **concatenated index**, is an index on multiple columns in a table.

Place columns in a composite index in the order that makes the most sense for the queries that will retrieve data. The columns need not be adjacent in the table.

Composite indexes can speed retrieval of data for `SELECT` statements in which the `WHERE` clause references all or the leading portion of the columns in the composite index. Therefore, the order of the columns used in the definition is important. In general, the most commonly accessed columns go first.

For example, suppose an application frequently queries the `last_name`, `job_id`, and `salary` columns in the `employees` table. Also assume that `last_name` has high cardinality, which means that the number of distinct values is large compared to the number of table rows. You create an index with the following column order:

```
CREATE INDEX employees_ix
   ON employees (last_name, job_id, salary);
```

Queries that access all three columns, only the `last_name` column, or only the `last_name` and `job_id` columns use this index. In this example, queries that do not access the `last_name` column do not use the index.

> **Note:**
>
> In some cases, such as when the leading column has very low cardinality, the database may use a skip scan of this index (see "Index Skip Scan").

Multiple indexes can exist on the same table with the same column order when they meet any of the following conditions:

- The indexes are of different types.

  For example, you can create bitmap and B-tree indexes on the same columns.

- The indexes use different partitioning schemes.

  For example, you can create indexes that are locally partitioned and indexes that are globally partitioned.

- The indexes have different uniqueness properties.

  For example, you can create both a unique and a non-unique index on the same set of columns.

For example, a nonpartitioned index, global partitioned index, and locally partitioned index can exist for the same table columns in the same order. Only one index with the same number of columns in the same order can be visible at any one time.

This capability enables you to migrate applications without the need to drop an existing index and re-create it with different attributes. Also, this capability is useful in an OLTP database when an index key keeps increasing, causing the database to insert new entries into the same set of index blocks. To alleviate such "hot spots," you could evolve the index from a nonpartitioned index into a global partitioned index.

If indexes on the same set of columns do not differ in type or partitioning scheme, then these indexes must use different column permutations. For example, the following SQL statements specify valid column permutations:

```
CREATE INDEX employee_idx1 ON employees (last_name, job_id);
CREATE INDEX employee_idx2 ON employees (job_id, last_name);
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* to learn more about creating multiple indexes on the same set of columns

## Unique and Nonunique Indexes

Indexes can be unique or nonunique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns.

For example, your application may require that no two employees have the same employee ID. In a unique index, one rowid exists for each data value. The data in the leaf blocks is sorted only by key.

Nonunique indexes permit duplicates values in the indexed column or columns. For example, the `first_name` column of the `employees` table may contain multiple `Mike` values. For a nonunique index, the rowid is included in the key in sorted order, so nonunique indexes are sorted by the index key and rowid (ascending).

Oracle Database does not index table rows in which all key columns are null, except for bitmap indexes or when the cluster key column value is null.

## Types of Indexes

Oracle Database provides several indexing schemes, which provide complementary performance functionality.

B-tree indexes are the standard index type. They are excellent for highly selective indexes (few rows correspond to each index entry) and primary key indexes. Used as concatenated indexes, a B-tree index can retrieve data sorted by the indexed columns. B-tree indexes have the subtypes shown in the following table.

**Table 5-1 B-Tree Index Subtypes**

| B-Tree Index Subtype | Description | To Learn More |
|---|---|---|
| Index-organized tables | An index-organized table differs from a heap-organized because the data is itself the index. | "Overview of Index-Organized Tables" |
| Reverse key indexes | In this type of index, the bytes of the index key are reversed, for example, 103 is stored as 301. The reversal of bytes spreads out inserts into the index over many blocks. | "Reverse Key Indexes" |
| Descending indexes | This type of index stores data on a particular column or columns in descending order. | "Ascending and Descending Indexes" |
| B-tree cluster indexes | This type of index stores data on a particular column or columns in descending order. | "Ascending and Descending Indexes" |

The following table shows types of indexes that do not use a B-tree structure.

**Table 5-2 Indexes Not Using a B-Tree Structure**

| Type | Description | To Learn More |
|---|---|---|
| Bitmap and bitmap join indexes | In a bitmap index, an index entry uses a bitmap to point to multiple rows. In contrast, a B-tree index entry points to a single row. A bitmap join index is a bitmap index for the join of two or more tables. | "Overview of Bitmap Indexes" |
| Function-based indexes | This type of index includes columns that are either transformed by a function, such as the `UPPER` function, or included in an expression. B-tree or bitmap indexes can be function-based. | "Overview of Function-Based Indexes" |
| Application domain indexes | A user creates this type of index for data in an application-specific domain. The physical index need not use a traditional index structure and can be stored either in the Oracle database as tables or externally as a file. | "Overview of Application Domain Indexes" |

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to manage indexes
> - *Oracle Database SQL Tuning Guide* to learn about different index access paths

## How the Database Maintains Indexes

The database automatically maintains and uses indexes after they are created.

Indexes automatically reflect data changes to their underlying tables. Examples of changes include adding, updating, and deleting rows. No user actions are required.

Retrieval performance of indexed data remains almost constant, even as rows are inserted. However, the presence of many indexes on a table degrades DML performance because the database must also update the indexes.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn more about automated indexing
> - *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

## Index Storage

Oracle Database stores index data in an index segment.

Space available for index data in a data block is the data block size minus block overhead, entry overhead, rowid, and one length byte for each value indexed.

The tablespace of an index segment is either the default tablespace of the owner or a tablespace specifically named in the `CREATE INDEX` statement. For ease of administration you can store an index in a separate tablespace from its table. For example, you may choose not to back up tablespaces containing only indexes, which can be rebuilt, and so decrease the time and storage required for backups.

> **See Also:**
>
> "Overview of Index Blocks" to learn about types of index block (root, branch, and leaf), and how index entries are stored within a block

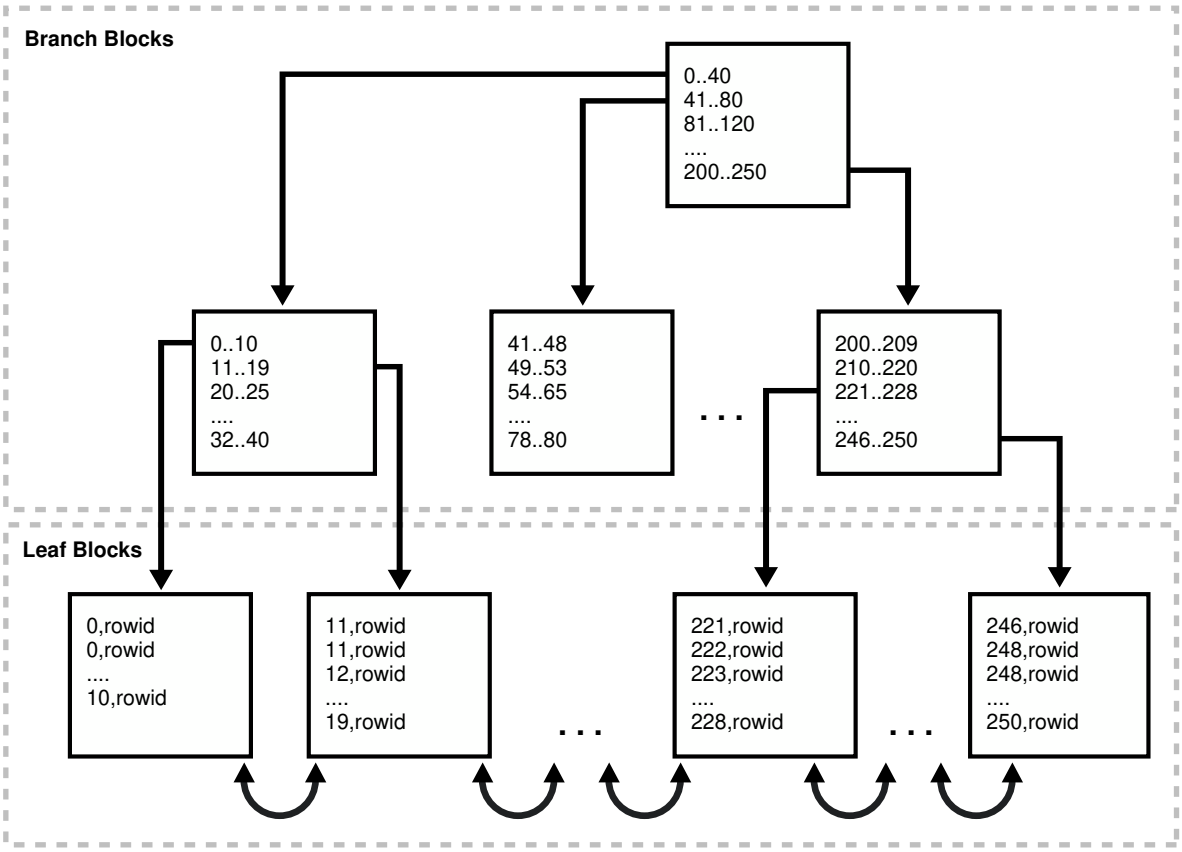
# Overview of B-Tree Indexes

B-trees, short for *balanced trees*, are the most common type of database index. A **B-tree index** is an ordered list of values divided into ranges. By associating a key with a row or range

of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.

The following figure illustrates the structure of a B-tree index. The example shows an index on the `department_id` column, which is a foreign key column in the `employees` table.

**Figure 5-1 Internal Structure of a B-tree Index**




This section contains the following topics:

- Branch Blocks and Leaf Blocks
- Index Scans
- Reverse Key Indexes
- Ascending and Descending Indexes
- Index Compression

- Branch Blocks and Leaf Blocks
  A B-tree index has two types of blocks: the **branch block** for searching, and the **leaf block** for storing key values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.
- Index Scans
  In an **index scan**, the database retrieves a row by traversing the index, using the indexed column values specified by the statement. If the database scans the index for a value, then it will find this value in *n* I/Os where *n* is the height of the B-tree index. This is the basic principle behind Oracle Database indexes.

- Reverse Key Indexes
  A **reverse key index** is a type of B-tree index that physically reverses the bytes of each index key while keeping the column order.
- Ascending and Descending Indexes
  In an **ascending index**, Oracle Database stores data in ascending order. By default, character data is ordered by the binary values contained in each byte of the value, numeric data from smallest to largest number, and date from earliest to latest value.
- Index Compression
  To reduce space in indexes, Oracle Database can employ different compression algorithms.

## Branch Blocks and Leaf Blocks

A B-tree index has two types of blocks: the **branch block** for searching, and the **leaf block** for storing key values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

In Figure 5-1, the root branch block has an entry `0-40`, which points to the leftmost block in the next branch level. This branch block contains entries such as `0-10` and `11-19`. Each of these entries points to a leaf block that contains key values that fall in the range.

A B-tree index is balanced because all leaf blocks automatically stay at the same depth. Thus, retrieval of any record from anywhere in the index takes approximately the same amount of time. The height of the index is the number of blocks required to go from the root block to a leaf block. The branch level is the height minus 1. In Figure 5-1, the index has a height of 3 and a branch level of 2.

Branch blocks store the minimum key prefix needed to make a branching decision between two keys. This technique enables the database to fit as much data as possible on each branch block. The branch blocks contain a pointer to the child block containing the key. The number of keys and pointers is limited by the block size.

The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. Each entry is sorted by (key, rowid). Within a leaf block, a key and rowid is linked to its left and right sibling entries. The leaf blocks themselves are also doubly linked. In Figure 5-1 the leftmost leaf block (`0-10`) is linked to the second leaf block (`11-19`).

> **Note:**
>
> Indexes in columns with character data are based on the binary values of the characters in the database character set.

## Index Scans

In an **index scan**, the database retrieves a row by traversing the index, using the indexed column values specified by the statement. If the database scans the index for a value, then it will find this value in *n* I/Os where *n* is the height of the B-tree index. This is the basic principle behind Oracle Database indexes.

If a SQL statement accesses only indexed columns, then the database reads values directly from the index rather than from the table. If the statement accesses nonindexed columns in addition to the indexed columns, then the database uses rowids to find the rows in the table.

Typically, the database retrieves table data by alternately reading an index block and then a table block.

- Full Index Scan
  In a **full index scan**, the database reads the entire index in order. A full index scan is available if a **predicate** (`WHERE` clause) in the SQL statement references a column in the index, and in some circumstances when no predicate is specified. A full scan can eliminate sorting because the data is ordered by index key.
- Fast Full Index Scan
  A **fast full index scan** is a full index scan in which the database accesses the data in the index itself without accessing the table, and the database reads the index blocks in no particular order.
- Index Range Scan
  An **index range scan** is an ordered scan of an index in which one or more leading columns of an index are specified in conditions, and 0, 1, or more values are possible for an index key.
- Index Unique Scan
  In contrast to an index range scan, an **index unique scan** must have either 0 or 1 rowid associated with an index key.
- Index Skip Scan
  An **index skip scan** uses logical subindexes of a composite index. The database "skips" through a single index as if it were searching separate indexes.
- Index Clustering Factor
  The **index clustering factor** measures row order in relation to an indexed value such as employee last name. As the degree of order increases, the clustering factor decreases.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for detailed information about index scans

## Full Index Scan

In a **full index scan**, the database reads the entire index in order. A full index scan is available if a **predicate** (`WHERE` clause) in the SQL statement references a column in the index, and in some circumstances when no predicate is specified. A full scan can eliminate sorting because the data is ordered by index key.

**Example 5-1 Full Index Scan**

Suppose that an application runs the following query:

```
SELECT department_id, last_name, salary
FROM   employees
WHERE  salary > 5000
ORDER BY department_id, last_name;
```

In this example, the `department_id`, `last_name`, and `salary` are a composite key in an index. Oracle Database performs a full scan of the index, reading it in sorted order (ordered by department ID and last name) and filtering on the salary attribute. In this way, the database scans a set of data smaller than the `employees` table, which contains more columns than are included in the query, and avoids sorting the data.

The full scan could read the index entries as follows:

```
50,Atkinson,2800,rowid
60,Austin,4800,rowid
70,Baer,10000,rowid
80,Abel,11000,rowid
80,Ande,6400,rowid
110,Austin,7200,rowid
.
.
.
```

## Fast Full Index Scan

A **fast full index scan** is a full index scan in which the database accesses the data in the index itself without accessing the table, and the database reads the index blocks in no particular order.

Fast full index scans are an alternative to a full table scan when both of the following conditions are met:

- The index must contain all columns needed for the query.
- A row containing all nulls must not appear in the query result set. For this result to be guaranteed, at least one column in the index must have either:
  - A `NOT NULL` constraint
  - A predicate applied to the column that prevents nulls from being considered in the query result set

**Example 5-2 Fast Full Index Scan**

Assume that an application issues the following query, which does not include an `ORDER BY` clause:

```
SELECT last_name, salary
FROM   employees;
```

The `last_name` column has a not null constraint. If the last name and salary are a composite key in an index, then a fast full index scan can read the index entries to obtain the requested information:

```
Baida,2900,rowid
Atkinson,2800,rowid
Zlotkey,10500,rowid
Austin,7200,rowid
Baer,10000,rowid
Austin,4800,rowid
.
.
.
```

## Index Range Scan

An **index range scan** is an ordered scan of an index in which one or more leading columns of an index are specified in conditions, and 0, 1, or more values are possible for an index key.

A condition specifies a combination of one or more expressions and logical (Boolean) operators. It returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

The database commonly uses an index range scan to access selective data. The selectivity is the percentage of rows in the table that the query selects, with 0 meaning no rows and 1 meaning all rows. Selectivity is tied to a query predicate, such as `WHERE last_name LIKE 'A%'`, or a combination of predicates. A predicate becomes more selective as the value approaches 0 and less selective (or more unselective) as the value approaches 1.

For example, a user queries employees whose last names begin with `A`. Assume that the `last_name` column is indexed, with entries as follows:

```
Abel,rowid
Ande,rowid
Atkinson,rowid
Austin,rowid
Austin,rowid
Baer,rowid
.
.
.
```

The database could use a range scan because the `last_name` column is specified in the predicate and multiples rowids are possible for each index key. For example, two employees are named Austin, so two rowids are associated with the key `Austin`.

An index range scan can be bounded on both sides, as in a query for departments with IDs between 10 and 40, or bounded on only one side, as in a query for IDs over 40. To scan the index, the database moves backward or forward through the leaf blocks. For example, a scan for IDs between 10 and 40 locates the first index leaf block that contains the lowest key value that is 10 or greater. The scan then proceeds horizontally through the linked list of leaf nodes until it locates a value greater than 40.

## Index Unique Scan

In contrast to an index range scan, an **index unique scan** must have either 0 or 1 rowid associated with an index key.

The database performs a unique scan when a predicate references all of the columns in the key of a `UNIQUE` index using an equality operator. An index unique scan stops processing as soon as it finds the first record because no second record is possible.

As an illustration, suppose that a user runs the following query:

```
SELECT *
FROM   employees
WHERE  employee_id = 5;
```

Assume that the `employee_id` column is the primary key and is indexed with entries as follows:

```
1,rowid
2,rowid
4,rowid
5,rowid
6,rowid
.
.
.
```

In this case, the database can use an index unique scan to locate the rowid for the employee whose ID is 5.

## Index Skip Scan

An **index skip scan** uses logical subindexes of a composite index. The database "skips" through a single index as if it were searching separate indexes.

Skip scanning is beneficial if there are few distinct values in the leading column of a composite index and many distinct values in the nonleading key of the index. The database may choose an index skip scan when the leading column of the composite index is not specified in a query predicate.

**Example 5-3 Skip Scan of a Composite Index**

Assume that you run the following query for a customer in the `sh.customers` table:

```
SELECT * FROM sh.customers WHERE cust_email = 'Abbey@company.example.com';
```

The `customers` table has a column `cust_gender` whose values are either `M` or `F`. Assume that a composite index exists on the columns (`cust_gender`, `cust_email`). The following example shows a portion of the index entries:

```
F,Wolf@company.example.com,rowid
F,Wolsey@company.example.com,rowid
F,Wood@company.example.com,rowid
F,Woodman@company.example.com,rowid
F,Yang@company.example.com,rowid
F,Zimmerman@company.example.com,rowid
M,Abbassi@company.example.com,rowid
M,Abbey@company.example.com,rowid
```

The database can use a skip scan of this index even though `cust_gender` is not specified in the `WHERE` clause.

In a skip scan, the number of logical subindexes is determined by the number of distinct values in the leading column. In the preceding example, the leading column has two possible values. The database logically splits the index into one subindex with the key `F` and a second subindex with the key `M`.

When searching for the record for the customer whose email is `Abbey@company.example.com`, the database searches the subindex with the value `F` first and then searches the subindex with the value `M`. Conceptually, the database processes the query as follows:

```
SELECT * FROM sh.customers WHERE cust_gender = 'F'
  AND cust_email = 'Abbey@company.example.com'
UNION ALL
SELECT * FROM sh.customers WHERE cust_gender = 'M'
  AND cust_email = 'Abbey@company.example.com';
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* to learn more about skip scans

## Index Clustering Factor

The **index clustering factor** measures row order in relation to an indexed value such as employee last name. As the degree of order increases, the clustering factor decreases.

The clustering factor is useful as a rough measure of the number of I/Os required to read an entire table using an index:

- If the clustering factor is high, then Oracle Database performs a relatively high number of I/Os during a large index range scan. The index entries point to random table blocks, so the database may have to read and reread the same blocks over and over again to retrieve the data pointed to by the index.
- If the clustering factor is low, then Oracle Database performs a relatively low number of I/Os during a large index range scan. The index keys in a range tend to point to the same data block, so the database does not have to read and reread the same blocks over and over.

The clustering factor is relevant for index scans because it can show:

- Whether the database will use an index for large range scans
- The degree of table organization in relation to the index key
- Whether you should consider using an index-organized table, partitioning, or table cluster if rows must be ordered by the index key

**Example 5-4 Clustering Factor**

Assume that the `employees` table fits into two data blocks. Table 5-3 depicts the rows in the two data blocks (the ellipses indicate data that is not shown).

**Table 5-3 Contents of Two Data Blocks in the Employees Table**

| Data Block 1 | Data Block 2 |
|---|---|
| 100 Steven **King** SKING ...<br>156 Janette **King** JKING ...<br>115 Alexander **Khoo** AKHOO ...<br>.<br>.<br>.<br>116 Shelli **Baida** SBAIDA ...<br>204 Hermann **Baer** HBAER ...<br>105 David **Austin** DAUSTIN ...<br>130 Mozhe **Atkinson** MATKINSO ...<br>166 Sundar **Ande** SANDE ...<br>174 Ellen **Abel** EABEL ... | 149 Eleni **Zlotkey** EZLOTKEY ...<br>200 Jennifer **Whalen** JWHALEN ...<br>.<br>.<br>.<br>137 Renske **Ladwig** RLADWIG ...<br>173 Sundita **Kumar** SKUMAR ...<br>101 Neena **Kochar** NKOCHHAR ... |

Rows are stored in the blocks in order of last name (shown in bold). For example, the bottom row in data block 1 describes Abel, the next row up describes Ande, and so on alphabetically

until the top row in block 1 for Steven King. The bottom row in block 2 describes Kochar, the next row up describes Kumar, and so on alphabetically until the last row in the block for Zlotkey.

Assume that an index exists on the last name column. Each name entry corresponds to a rowid. Conceptually, the index entries would look as follows:

```
Abel,block1row1
Ande,block1row2
Atkinson,block1row3
Austin,block1row4
Baer,block1row5
.
.
.
```

Assume that a separate index exists on the employee ID column. Conceptually, the index entries might look as follows, with employee IDs distributed in almost random locations throughout the two blocks:

```
100,block1row50
101,block2row1
102,block1row9
103,block2row19
104,block2row39
105,block1row4
.
.
.
```

The following statement queries the `ALL_INDEXES` view for the clustering factor for these two indexes:

```
SQL> SELECT INDEX_NAME, CLUSTERING_FACTOR
  2  FROM ALL_INDEXES
  3  WHERE INDEX_NAME IN ('EMP_NAME_IX','EMP_EMP_ID_PK');

INDEX_NAME           CLUSTERING_FACTOR
-------------------- -----------------
EMP_EMP_ID_PK                       19
EMP_NAME_IX                          2
```

The clustering factor for `EMP_NAME_IX` is low, which means that adjacent index entries in a single leaf block tend to point to rows in the same data blocks. The clustering factor for `EMP_EMP_ID_PK` is high, which means that adjacent index entries in the same leaf block are much less likely to point to rows in the same data blocks.

> **See Also:**
>
> *Oracle Database Reference* to learn about `ALL_INDEXES`

## Reverse Key Indexes

A **reverse key index** is a type of B-tree index that physically reverses the bytes of each index key while keeping the column order.

For example, if the index key is `20`, and if the two bytes stored for this key in hexadecimal are `C1,15` in a standard B-tree index, then a reverse key index stores the bytes as `15,C1`.

Reversing the key solves the problem of contention for leaf blocks in the right side of a B-tree index. This problem can be especially acute in an Oracle Real Application Clusters (Oracle RAC) database in which multiple instances repeatedly modify the same block. For example, in an `orders` table the primary keys for orders are sequential. One instance in the cluster adds order 20, while another adds 21, with each instance writing its key to the same leaf block on the right-hand side of the index.

In a reverse key index, the reversal of the byte order distributes inserts across all leaf keys in the index. For example, keys such as 20 and 21 that would have been adjacent in a standard key index are now stored far apart in separate blocks. Thus, I/O for insertions of sequential keys is more evenly distributed.

Because the data in the index is not sorted by column key when it is stored, the reverse key arrangement eliminates the ability to run an index range scanning query in some cases. For example, if a user issues a query for order IDs greater than 20, then the database cannot start with the block containing this ID and proceed horizontally through the leaf blocks.

## Ascending and Descending Indexes

In an **ascending index**, Oracle Database stores data in ascending order. By default, character data is ordered by the binary values contained in each byte of the value, numeric data from smallest to largest number, and date from earliest to latest value.

For an example of an ascending index, consider the following SQL statement:

```
CREATE INDEX emp_deptid_ix ON hr.employees(department_id);
```

Oracle Database sorts the `hr.employees` table on the `department_id` column. It loads the ascending index with the `department_id` and corresponding rowid values in ascending order, starting with `0`. When it uses the index, Oracle Database searches the sorted `department_id` values and uses the associated rowids to locate rows having the requested `department_id` value.

By specifying the `DESC` keyword in the `CREATE INDEX` statement, you can create a descending index. In this case, the index stores data on a specified column or columns in descending order. If the index in Table 5-3 on the `employees.department_id` column were descending, then the leaf blocking containing `250` would be on the left side of the tree and block with `0` on the right. The default search through a descending index is from highest to lowest value.

Descending indexes are useful when a query sorts some columns ascending and others descending. For an example, assume that you create a composite index on the `last_name` and `department_id` columns as follows:

```
CREATE INDEX emp_name_dpt_ix ON hr.employees(last_name ASC, department_id
DESC);
```

If a user queries `hr.employees` for last names in ascending order (A to Z) and department IDs in descending order (high to low), then the database can use this index to retrieve the data and avoid the extra step of sorting it.

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* to learn more about ascending and descending index searches
> - *Oracle Database SQL Language Reference* for descriptions of the `ASC` and `DESC` options of `CREATE INDEX`

# Index Compression

To reduce space in indexes, Oracle Database can employ different compression algorithms.

- Prefix Compression
  Oracle Database can use **prefix compression**, also known as **key compression**, to compress portions of the primary key column values in a B-tree index or an index-organized table. Prefix compression can greatly reduce the space consumed by the index.
- Advanced Index Compression
  Starting with Oracle Database 12c Release 1 (12.1.0.2), **advanced index compression** improves on traditional prefix compression for supported indexes on heap-organized tables.

## Prefix Compression

Oracle Database can use **prefix compression**, also known as **key compression**, to compress portions of the primary key column values in a B-tree index or an index-organized table. Prefix compression can greatly reduce the space consumed by the index.

An uncompressed index entry has one piece. An index entry using prefix compression has two pieces: a prefix entry, which is the grouping piece, and a suffix entry, which is the unique or nearly unique piece. The database achieves compression by sharing the prefix entries among the suffix entries in an index block.

> **Note:**
>
> If a key is not defined to have a unique piece, then the database provides one by appending a rowid to the grouping piece.

By default, the prefix of a unique index consists of all key columns excluding the last one, whereas the prefix of a nonunique index consists of all key columns. Suppose you create a composite, unique index on two columns of the `oe.orders` table as follows:

```
CREATE UNIQUE INDEX orders_mod_stat_ix ON orders ( order_mode, order_status );
```

In the preceding example, an index key might be `online,0`. The rowid is stored in the key data portion of the entry, and is not part of the key itself.

> **Note:**
>
> If you create a unique index on a single column, then Oracle Database cannot use prefix key compression because no common prefixes exist.

Alternatively, suppose you create a nonunique index on the same columns:

```
CREATE INDEX orders_mod_stat_ix ON orders ( order_mode, order_status );
```

Also assume that repeated values occur in the `order_mode` and `order_status` columns. An index block could have entries as shown in the follow example:

```
online,0,AAAPvCAAFAAAAFaAAa
online,0,AAAPvCAAFAAAAFaAAg
online,0,AAAPvCAAFAAAAFaAAl
online,2,AAAPvCAAFAAAAFaAAm
online,3,AAAPvCAAFAAAAFaAAq
online,3,AAAPvCAAFAAAAFaAAt
```

In the preceding example, the key prefix would consist of a concatenation of the `order_mode` and `order_status` values, as in `online,0`. The suffix consists in the rowid, as in `AAAPvCAAFAAAAFaAAa`. The rowid makes the whole index entry unique because a rowid is itself unique in the database.

If the index in the preceding example were created with default prefix compression (specified by the `COMPRESS` keyword), then duplicate key prefixes such as `online,0` and `online,3` would be compressed. Conceptually, the database achieves compression as follows:

```
online,0
AAAPvCAAFAAAAFaAAa
AAAPvCAAFAAAAFaAAg
AAAPvCAAFAAAAFaAAl
online,2
AAAPvCAAFAAAAFaAAm
online,3
AAAPvCAAFAAAAFaAAq
AAAPvCAAFAAAAFaAAt
```

Suffix entries (the rowids) form the compressed version of index rows. Each suffix entry references a prefix entry, which is stored in the same index block as the suffix.

Alternatively, you could specify a prefix length when creating an index that uses prefix compression. For example, if you specified `COMPRESS 1`, then the prefix would be `order_mode` and the suffix would be `order_status,rowid`. For the values in the index block example, the index would factor out duplicate occurrences of the prefix `online`, which can be represented conceptually as follows:

```
online
0,AAAPvCAAFAAAAFaAAa
0,AAAPvCAAFAAAAFaAAg
0,AAAPvCAAFAAAAFaAAl
2,AAAPvCAAFAAAAFaAAm
3,AAAPvCAAFAAAAFaAAq
3,AAAPvCAAFAAAAFaAAt
```

The index stores a specific prefix once per leaf block at most. Only keys in the leaf blocks of a B-tree index are compressed. In the branch blocks the key suffix can be truncated, but the key is not compressed.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to use compressed indexes
- *Oracle Database VLDB and Partitioning Guide* to learn how to use prefix compression for partitioned indexes
- *Oracle Database SQL Language Reference* for descriptions of the *key_compression* clause of `CREATE INDEX`

## Advanced Index Compression

Starting with Oracle Database 12c Release 1 (12.1.0.2), **advanced index compression** improves on traditional prefix compression for supported indexes on heap-organized tables.

### Benefits of Advanced Index Compression

Prefix compression has limitations for types of indexes supported, compression ratio, and ease of use. Unlike prefix compression, which uses fixed duplicate key elimination for every block, advanced index compression uses adaptive duplicate key elimination on a per-block basis. The main advantages of advanced index compression are:

- The database automatically chooses the best compression for each block, using a number of internal algorithms such as intra-column level prefixes, duplicate key elimination, and rowid compression. Unlike in prefix compression, advanced index compression does not require the user to know data characteristics.
- Advanced compression works on both non-unique *and* unique indexes. Prefix compression works well on some non-unique indexes, but the ratios are lower on indexes whose leading columns do not have many repeats.
- The compressed index is usable in the same way as an uncompressed index. The index supports the same access paths: unique key lookups, range scans, and fast full scans.
- Indexes can inherit advanced compression from a parent table or containing tablespace.

### How Advanced Index Compression Works

Advanced index compression works at the block level to provide the best compression for each block. The database uses the following technique:

- During index creation, as a leaf block becomes full, the database automatically compresses the block to the optimal level.
- When reorganizing an index block as a result of DML, if the database can create sufficient space for the incoming index entry, then a block split does not occur. During DML without advanced index compression, however, an index block split always occurs when the block becomes full.

### Advanced Index Compression HIGH

In releases previous to Oracle Database 12c Release 2 (12.2), the only form of advanced index compression was low compression (`COMPRESS ADVANCED LOW`). Now you can also specify high compression (`COMPRESS ADVANCED HIGH`), which is the default. Advanced index compression with the `HIGH` option offers the following advantages:

- Gives higher compression ratios in most cases, while also improving performance for queries that access the index
- Employs more complex compression algorithms than advanced low
- Stores data in a compression unit, which is a special on-disk format

> **Note:**
>
> When you apply `HIGH` compression, all blocks have compression. When you apply `LOW` compression, the database may leave some blocks uncompressed. You can use statistics to determine how many blocks were left uncompressed.

**Example 5-5 Creating an Index with Advanced High Compression**

This example enables advanced index compression for an index on the `hr.employees` table:

```
CREATE INDEX hr.emp_mndp_ix
  ON hr.employees(manager_id, department_id)
  COMPRESS ADVANCED;
```

The following query shows the type of compression:

```
SELECT COMPRESSION FROM DBA_INDEXES WHERE INDEX_NAME ='EMP_MNDP_IX';

COMPRESSION
-------------
ADVANCED HIGH
```

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to enable advanced index compression
> - *Oracle Database SQL Language Reference*for descriptions of the *key_compression* clause of `CREATE INDEX`
> - *Oracle Database Reference* to learn about `ALL_INDEXES`

# Overview of Bitmap Indexes

In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows.

Bitmap indexes are primarily designed for data warehousing or environments in which queries reference many columns in an ad hoc fashion. Situations that may call for a bitmap index include:

- The indexed columns have low cardinality, that is, the number of distinct values is small compared to the number of table rows.
- The indexed table is either read-only or not subject to significant modification by DML statements.

For a data warehouse example, the `sh.customers` table has a `cust_gender` column with only two possible values: `M` and `F`. Suppose that queries for the number of customers of a particular gender are common. In this case, the `customers.cust_gender` column would be a candidate for a bitmap index.

Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a B-tree index although it uses a different internal representation.

If the indexed column in a single row is updated, then the database locks the index key entry (for example, `M` or `F`) and not the individual bit mapped to the updated row. Because a key points to many rows, DML on indexed data typically locks all of these rows. For this reason, bitmap indexes are not appropriate for many OLTP applications.

- Example: Bitmap Indexes on a Single Table
  In this example, some columns of `sh.customers` table are candidates for a bitmap index.
- Bitmap Join Indexes
  A **bitmap join index** is a bitmap index for the **join** of two or more tables.
- Bitmap Storage Structure
  Oracle Database uses a B-tree index structure to store bitmaps for each indexed key.

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* to learn more about bitmap indexes
> - *Oracle Database Data Warehousing Guide* to learn how to use bitmap indexes in a data warehouse

## Example: Bitmap Indexes on a Single Table

In this example, some columns of `sh.customers` table are candidates for a bitmap index.

Consider the following query:

```
SQL> SELECT cust_id, cust_last_name, cust_marital_status, cust_gender
  2  FROM   sh.customers
  3  WHERE  ROWNUM < 8 ORDER BY cust_id;

   CUST_ID CUST_LAST_ CUST_MAR C
---------- ---------- -------- -
         1 Kessel              M
         2 Koch                F
         3 Emmerson            M
         4 Hardy               M
         5 Gowen               M
         6 Charles    single   F
```

```
         7 Ingram     single    F

7 rows selected.
```

The `cust_marital_status` and `cust_gender` columns have low cardinality, whereas `cust_id` and `cust_last_name` do not. Thus, bitmap indexes may be appropriate on `cust_marital_status` and `cust_gender`. A bitmap index is probably not useful for the other columns. Instead, a unique B-tree index on these columns would likely provide the most efficient representation and retrieval.

Table 5-4 illustrates the bitmap index for the `cust_gender` column output shown in the preceding example. It consists of two separate bitmaps, one for each gender.

**Table 5-4 Sample Bitmap for One Column**

| Value | Row 1 | Row 2 | Row 3 | Row 4 | Row 5 | Row 6 | Row 7 |
|---|---|---|---|---|---|---|---|
| M | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| F | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

A mapping function converts each bit in the bitmap to a rowid of the `customers` table. Each bit value depends on the values of the corresponding row in the table. For example, the bitmap for the `M` value contains a `1` as its first bit because the gender is `M` in the first row of the `customers` table. The bitmap `cust_gender='M'` has a `0` for the bits in rows 2, 6, and 7 because these rows do not contain `M` as their value.

> **Note:**
>
> Bitmap indexes can include keys that consist entirely of null values, unlike B-tree indexes. Indexing nulls can be useful for some SQL statements, such as queries with the aggregate function `COUNT`.

An analyst investigating demographic trends of the customers may ask, "How many of our female customers are single or divorced?" This question corresponds to the following SQL query:

```
SELECT COUNT(*)
FROM   customers
WHERE  cust_gender = 'F'
AND    cust_marital_status IN ('single', 'divorced');
```

Bitmap indexes can process this query efficiently by counting the number of `1` values in the resulting bitmap, as illustrated in Table 5-5. To identify the customers who satisfy the criteria, Oracle Database can use the resulting bitmap to access the table.

**Table 5-5 Sample Bitmap for Two Columns**

| Value | Row 1 | Row 2 | Row 3 | Row 4 | Row 5 | Row 6 | Row 7 |
|---|---|---|---|---|---|---|---|
| M | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| F | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

**Table 5-5 (Cont.) Sample Bitmap for Two Columns**

| Value | Row 1 | Row 2 | Row 3 | Row 4 | Row 5 | Row 6 | Row 7 |
|---|---|---|---|---|---|---|---|
| `single` | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| `divorced` | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| `single` or `divorced`, and `F` | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Bitmap indexing efficiently merges indexes that correspond to several conditions in a `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This technique improves response time, often dramatically.

## Bitmap Join Indexes

A **bitmap join index** is a bitmap index for the **join** of two or more tables.

For each value in a table column, the index stores the rowid of the corresponding row in the indexed table. In contrast, a standard bitmap index is created on a single table.

A bitmap join index is an efficient means of reducing the volume of data that must be joined by performing restrictions in advance. For an example of when a bitmap join index would be useful, assume that users often query the number of employees with a particular job type. A typical query might look as follows:

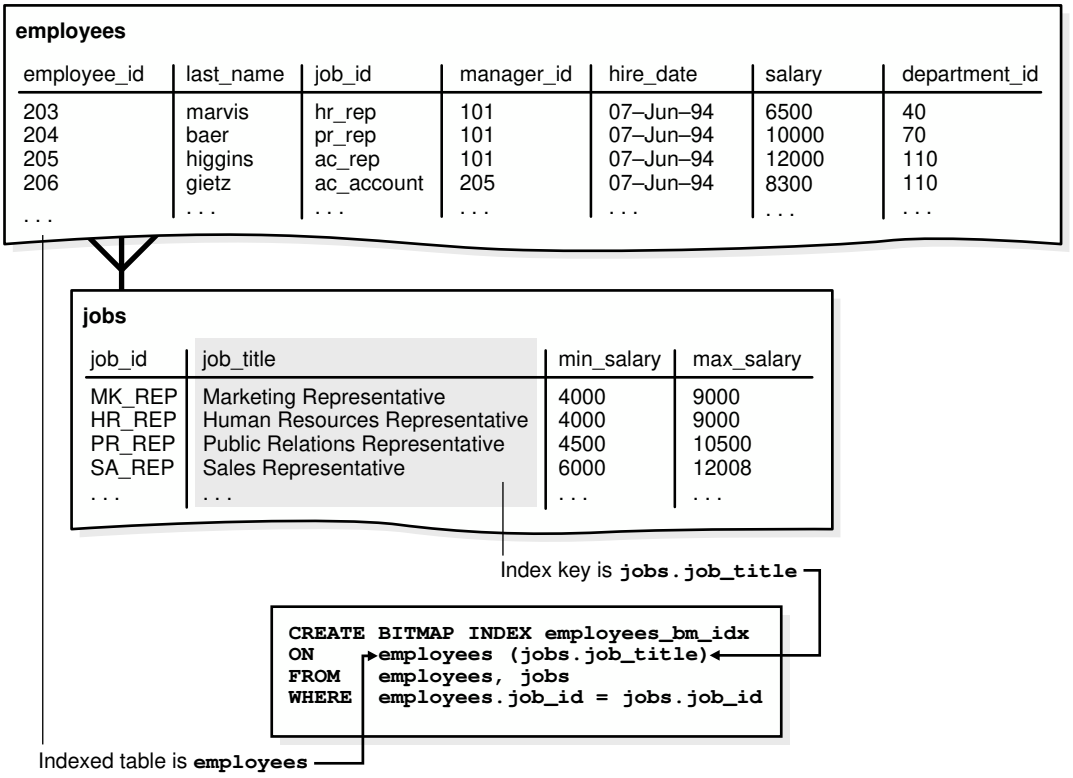
```
SELECT COUNT(*)
FROM   employees, jobs
WHERE  employees.job_id = jobs.job_id
AND    jobs.job_title = 'Accountant';
```

The preceding query would typically use an index on `jobs.job_title` to retrieve the rows for `Accountant` and then the job ID, and an index on `employees.job_id` to find the matching rows. To retrieve the data from the index itself rather than from a scan of the tables, you could create a bitmap join index as follows:

```
CREATE BITMAP INDEX employees_bm_idx
ON     employees (jobs.job_title)
FROM   employees, jobs
WHERE  employees.job_id = jobs.job_id;
```

As illustrated in the following figure, the index key is `jobs.job_title` and the indexed table is `employees`.

**Figure 5-2 Bitmap Join Index**




Conceptually, `employees_bm_idx` is an index of the `jobs.title` column in the SQL query shown in the following query (sample output included). The `job_title` key in the index points to rows in the `employees` table. A query of the number of accountants can use the index to avoid accessing the `employees` and `jobs` tables because the index itself contains the requested information.

```
SELECT jobs.job_title AS "jobs.job_title", employees.rowid AS
"employees.rowid"
FROM   employees, jobs
WHERE  employees.job_id = jobs.job_id
ORDER BY job_title;

jobs.job_title                     employees.rowid
---------------------------------- ------------------
Accountant                         AAAQNKAAFAAAABSAAL
Accountant                         AAAQNKAAFAAAABSAAN
Accountant                         AAAQNKAAFAAAABSAAM
Accountant                         AAAQNKAAFAAAABSAAJ
Accountant                         AAAQNKAAFAAAABSAAK
Accounting Manager                 AAAQNKAAFAAAABTAAH
Administration Assistant           AAAQNKAAFAAAABTAAC
Administration Vice President      AAAQNKAAFAAAABSAAC
Administration Vice President      AAAQNKAAFAAAABSAAB
.
.
.
```

In a data warehouse, the join condition is an equijoin (it uses the equality operator) between the primary key columns of the dimension tables and the foreign key columns in the fact table. Bitmap join indexes are sometimes much more efficient in storage than materialized join views, an alternative for materializing joins in advance.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on bitmap join indexes

## Bitmap Storage Structure

Oracle Database uses a B-tree index structure to store bitmaps for each indexed key.

For example, if `jobs.job_title` is the key column of a bitmap index, then one B-tree stores the index data. The leaf blocks store the individual bitmaps.

**Example 5-6 Bitmap Storage Example**

Assume that the `jobs.job_title` column has unique values `Shipping Clerk`, `Stock Clerk`, and several others. A bitmap index entry for this index has the following components:

- The job title as the index key
- A low rowid and high rowid for a range of rowids
- A bitmap for specific rowids in the range

Conceptually, an index leaf block in this index could contain entries as follows:

```
Shipping Clerk,AAAPzRAAFAAAABSABQ,AAAPzRAAFAAAABSABZ,0010000100
Shipping Clerk,AAAPzRAAFAAAABSABa,AAAPzRAAFAAAABSABh,010010
Stock Clerk,AAAPzRAAFAAAABSAAa,AAAPzRAAFAAAABSAAc,1001001100
Stock Clerk,AAAPzRAAFAAAABSAAd,AAAPzRAAFAAAABSAAt,0101001001
Stock Clerk,AAAPzRAAFAAAABSAAu,AAAPzRAAFAAAABSABz,100001
.
.
.
```

The same job title appears in multiple entries because the rowid range differs.

A session updates the job ID of one employee from `Shipping Clerk` to `Stock Clerk`. In this case, the session requires exclusive access to the index key entry for the old value (`Shipping Clerk`) and the new value (`Stock Clerk`). Oracle Database locks the rows pointed to by these two entries—but not the rows pointed to by `Accountant` or any other key—until the `UPDATE` commits.

The data for a bitmap index is stored in one segment. Oracle Database stores each bitmap in one or more pieces. Each piece occupies part of a single data block.

> **See Also:**
>
> "User Segments" explains the different types of segments, and how segments are created

# Overview of Function-Based Indexes

A **function-based index** computes the value of a function or expression involving one or more columns and stores it in an index. A function-based index can be either a B-tree or a bitmap index.

The indexed function can be an arithmetic expression or an expression that contains a SQL function, user-defined PL/SQL function, package function, or C callout. For example, a function could add the values in two columns.

- Uses of Function-Based Indexes
  Function-based indexes are efficient for evaluating statements that contain functions in their `WHERE` clauses. The database only uses the function-based index when the function is included in a query. When the database processes `INSERT` and `UPDATE` statements, however, it must still evaluate the function to process the statement.
- Optimization with Function-Based Indexes
  For queries with expressions in a `WHERE` clause, the optimizer might use an index range scan on a function-based index.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to create function-based indexes
> - *Oracle Database Development Guide* for more information about using function-based indexes
> - *Oracle Database SQL Language Reference*for restrictions and usage notes for function-based indexes

## Uses of Function-Based Indexes

Function-based indexes are efficient for evaluating statements that contain functions in their `WHERE` clauses. The database only uses the function-based index when the function is included in a query. When the database processes `INSERT` and `UPDATE` statements, however, it must still evaluate the function to process the statement.

**Example 5-7 Index Based on Arithmetic Expression**

For example, suppose you create the following function-based index:

```
CREATE INDEX emp_total_sal_idx
  ON employees (12 * salary * commission_pct, salary, commission_pct);
```

The database can use the preceding index when processing queries such as the following (partial sample output included):

```
SELECT   employee_id, last_name, first_name,
         12*salary*commission_pct AS "ANNUAL SAL"
FROM     employees
WHERE    (12 * salary * commission_pct) < 30000
ORDER BY "ANNUAL SAL" DESC;

EMPLOYEE_ID LAST_NAME                 FIRST_NAME           ANNUAL SAL
----------- ------------------------- -------------------- ----------
        159 Smith                     Lindsey                   28800
        151 Bernstein                 David                     28500
        152 Hall                      Peter                     27000
        160 Doran                     Louise                    27000
        175 Hutton                    Alyssa                    26400
        149 Zlotkey                   Eleni                     25200
        169 Bloom                     Harrison                  24000
```

**Example 5-8 Index Based on an UPPER Function**

Function-based indexes defined on the SQL functions `UPPER(column_name)` or `LOWER(column_name)` facilitate case-insensitive searches. For example, suppose that the `first_name` column in `employees` contains mixed-case characters. You create the following function-based index on the `hr.employees` table:

```
CREATE INDEX emp_fname_uppercase_idx
ON employees ( UPPER(first_name) );
```

The `emp_fname_uppercase_idx` index can facilitate queries such as the following:

```
SELECT *
FROM   employees
WHERE  UPPER(first_name) = 'AUDREY';
```

**Example 5-9 Indexing Specific Rows in a Table**

A function-based index is also useful for indexing only specific rows in a table. For example, the `cust_valid` column in the `sh.customers` table has either `I` or `A` as a value. To index only the `A` rows, you could write a function that returns a null value for any rows other than the `A` rows. You could create the index as follows:

```
CREATE INDEX cust_valid_idx
ON customers ( CASE cust_valid WHEN 'A' THEN 'A' END );
```

**See Also:**

- *Oracle Database Globalization Support Guide* for information about linguistic indexes
- *Oracle Database SQL Language Reference* to learn more about SQL functions

## Optimization with Function-Based Indexes

For queries with expressions in a `WHERE` clause, the optimizer might use an index range scan on a function-based index.

The range scan access path is especially beneficial when the predicate is highly selective, that is, when it chooses relatively few rows. In Example 5-7, if an index is built on the expression `12*salary*commission_pct`, then the optimizer can use an index range scan.

A virtual column is also useful for speeding access to data derived from expressions. For example, you could define virtual column `annual_sal` as `12*salary*commission_pct` and create a function-based index on `annual_sal`.

The optimizer performs expression matching by parsing the expression in a SQL statement and then comparing the expression trees of the statement and the function-based index. This comparison is case-insensitive and ignores blank spaces.

Different queries may use different types of index scans.

**See Also:**

- "Overview of the Optimizer"
- *Oracle Database SQL Tuning Guide* to learn more about gathering statistics
- *Oracle Database Administrator's Guide* to learn how to add virtual columns to a table

# Overview of Application Domain Indexes

An **application domain index** is a customized index specific to an application.

Extensive indexing can:

- Accommodate indexes on customized, complex data types such as documents, spatial data, images, and video clips (see )
- Make use of specialized indexing techniques

You can encapsulate application-specific index management routines as an indextype schema object, and then define a domain index on table columns or attributes of an object type. Extensible indexing can efficiently process application-specific operators.

The application software, called the *cartridge*, controls the structure and content of a domain index. The database interacts with the application to build, maintain, and search the domain index. The index structure itself can be stored in the database as an index-organized table or externally as a file.

**See Also:**

- *Oracle Database Development Guide* to learn more about complex data types
- *Oracle Database Data Cartridge Developer's Guide* for information about using data cartridges within the Oracle Database extensibility architecture

# Overview of Index-Organized Tables

An **index-organized table** is a table stored in a variation of a B-tree index structure. In contrast, a **heap-organized table** inserts rows where they fit.

In an index-organized table, rows are stored in an index defined on the primary key for the table. Each index entry in the B-tree also stores the non-key column values. Thus, the index is the data, and the data is the index. Applications manipulate index-organized tables just like heap-organized tables, using SQL statements.

For an analogy of an index-organized table, suppose a human resources manager has a book case of cardboard boxes. Each box is labeled with a number—1, 2, 3, 4, and so on—but the boxes do not sit on the shelves in sequential order. Instead, each box contains a pointer to the shelf location of the next box in the sequence.

Folders containing employee records are stored in each box. The folders are sorted by employee ID. Employee King has ID 100, which is the lowest ID, so his folder is at the bottom of box 1. The folder for employee 101 is on top of 100, 102 is on top of 101, and so on until box 1 is full. The next folder in the sequence is at the bottom of box 2.

In this analogy, ordering folders by employee ID makes it possible to search efficiently for folders without having to maintain a separate index. Suppose a user requests the records for employees 107, 120, and 122. Instead of searching an index in one step and retrieving the folders in a separate step, the manager can search the folders in sequential order and retrieve each folder as found.

Index-organized tables provide faster access to table rows by primary key or a valid prefix of the key. The presence of non-key columns of a row in the leaf block avoids an additional data block I/O. For example, the salary of employee 100 is stored in the index row itself. Also, because rows are stored in primary key order, range access by the primary key or prefix involves minimal block I/Os. Another benefit is the avoidance of the space overhead of a separate primary key index.

Index-organized tables are useful when related pieces of data must be stored together or data must be physically stored in a specific order. A typical use of this type of table is for information retrieval, spatial data, and OLAP applications.

- Index-Organized Table Characteristics
  The database system performs all operations on index-organized tables by manipulating the B-tree index structure.
- Index-Organized Tables with Row Overflow Area
  When creating an index-organized table, you can specify a separate segment as a row overflow area.
- Secondary Indexes on Index-Organized Tables
  A **secondary index** is an index on an index-organized table.

> **See Also:**
>
> - *Oracle Spatial and Graph Developer's Guide* for an overview of Oracle Spatial and Graph
> - *Oracle OLAP User's Guide* to learn about OLAP Technology in the Oracle Database
> - *Oracle Database Administrator's Guide* to learn how to manage index-organized tables
> - *Oracle Database SQL Language Reference* for `CREATE TABLE ... ORGANIZATION INDEX` syntax and semantics

## Index-Organized Table Characteristics

The database system performs all operations on index-organized tables by manipulating the B-tree index structure.

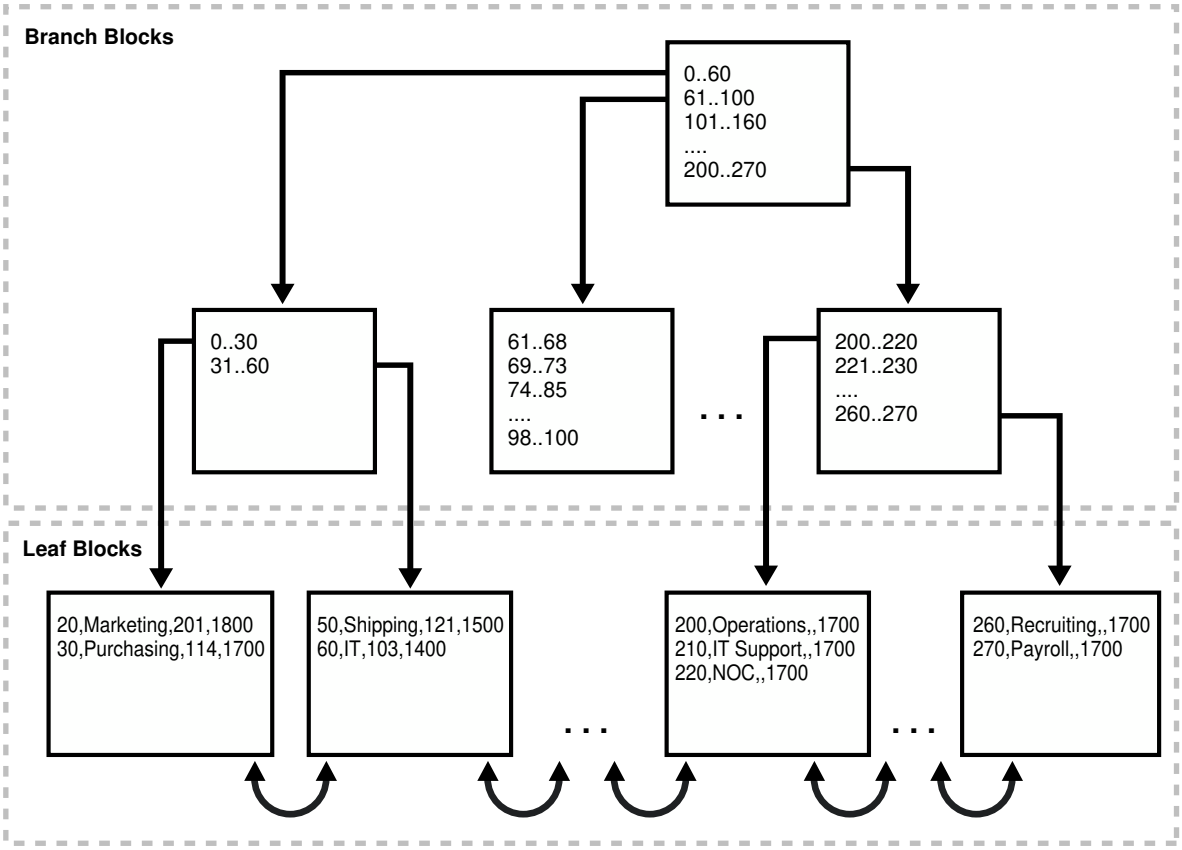
The following table summarizes the differences between index-organized tables and heap-organized tables.

**Table 5-6 Comparison of Heap-Organized Tables with Index-Organized Tables**

| Heap-Organized Table | Index-Organized Table |
|---|---|
| The rowid uniquely identifies a row. Primary key constraint may optionally be defined. | Primary key uniquely identifies a row. Primary key constraint must be defined. |
| Physical rowid in `ROWID` pseudocolumn allows building secondary indexes. | Logical rowid in `ROWID` pseudocolumn allows building secondary indexes. |
| Individual rows may be accessed directly by rowid. | Access to individual rows may be achieved indirectly by primary key. |
| Sequential full table scan returns all rows in some order. | A full index scan or fast full index scan returns all rows in some order. |
| Can be stored in a table cluster with other tables. | Cannot be stored in a table cluster. |
| Can contain a column of the `LONG` data type and columns of LOB data types. | Can contain LOB columns but not `LONG` columns. |
| Can contain virtual columns (only relational heap tables are supported). | Cannot contain virtual columns. |

Figure 5-3 illustrates the structure of an index-organized `departments` table. The leaf blocks contain the rows of the table, ordered sequentially by primary key. For example, the first value in the first leaf block shows a department ID of `20`, department name of `Marketing`, manager ID of `201`, and location ID of `1800`.

**Figure 5-3 Index-Organized Table**




### Example 5-10 Scan of Index-Organized Table

An index-organized table stores all data in the same structure and does not need to store the rowid. As shown in Figure 5-3, leaf block 1 in an index-organized table might contain entries as follows, ordered by primary key:

```
20,Marketing,201,1800
30,Purchasing,114,1700
```

Leaf block 2 in an index-organized table might contain entries as follows:

```
50,Shipping,121,1500
60,IT,103,1400
```

A scan of the index-organized table rows in primary key order reads the blocks in the following sequence:

1. Block 1
2. Block 2

**Example 5-11 Scan of Heap-Organized Table**

To contrast data access in a heap-organized table to an index-organized table, suppose block 1 of a heap-organized `departments` table segment contains rows as follows:

```
50,Shipping,121,1500
20,Marketing,201,1800
```

Block 2 contains rows for the same table as follows:

```
30,Purchasing,114,1700
60,IT,103,1400
```

A B-tree index leaf block for this heap-organized table contains the following entries, where the first value is the primary key and the second is the rowid:

```
20,AAAPeXAAFAAAAAyAAD
30,AAAPeXAAFAAAAAyAAA
50,AAAPeXAAFAAAAAyAAC
60,AAAPeXAAFAAAAAyAAB
```

A scan of the table rows in primary key order reads the table segment blocks in the following sequence:

1. Block 1
2. Block 2
3. Block 1
4. Block 2

Thus, the number of block I/Os in this example is double the number in the index-organized example.

> **See Also:**
>
> - "Table Organization"
>
>  to learn more about heap-organized tables
> - "Introduction to Logical Storage Structures"
>
>  to learn more about the relationship between segments and data blocks

## Index-Organized Tables with Row Overflow Area

When creating an index-organized table, you can specify a separate segment as a row overflow area.

In index-organized tables, B-tree index entries can be large because they contain an entire row, so a separate segment to contain the entries is useful. In contrast, B-tree entries are usually small because they consist of the key and rowid.

If a row overflow area is specified, then the database can divide a row in an index-organized table into the following parts:

- The index entry

  This part contains column values for all the primary key columns, a physical rowid that points to the overflow part of the row, and optionally a few of the non-key columns. This part is stored in the index segment.

- The overflow part

  This part contains column values for the remaining non-key columns. This part is stored in the overflow storage area segment.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to use the `OVERFLOW` clause of `CREATE TABLE` to set a row overflow area
> - *Oracle Database SQL Language Reference*for `CREATE TABLE ... OVERFLOW` syntax and semantics

## Secondary Indexes on Index-Organized Tables

A **secondary index** is an index on an index-organized table.

In a sense, a secondary index is an index on an index. It is an independent schema object and is stored separately from the index-organized table.

Oracle Database uses row identifiers called logical rowids for index-organized tables. A logical rowid is a base64-encoded representation of the table primary key. The logical rowid length depends on the primary key length.

Rows in index leaf blocks can move within or between blocks because of insertions. Rows in index-organized tables do not migrate as heap-organized rows do. Because rows in index-organized tables do not have permanent physical addresses, the database uses logical rowids based on primary key.

For example, assume that the `departments` table is index-organized. The `location_id` column stores the ID of each department. The table stores rows as follows, with the last value as the location ID:

```
10,Administration,200,1700
20,Marketing,201,1800
30,Purchasing,114,1700
40,Human Resources,203,2400
```

A secondary index on the `location_id` column might have index entries as follows, where the value following the comma is the logical rowid:

```
1700,*BAFAJqoCwR/+
1700,*BAFAJqoCwQv+
1800,*BAFAJqoCwRX+
2400,*BAFAJqoCwSn+
```

Secondary indexes provide fast and efficient access to index-organized tables using columns that are neither the primary key nor a prefix of the primary key. For example, a query of the names of departments whose ID is greater than 1700 could use the secondary index to speed data access.

- Logical Rowids and Physical Guesses
  Secondary indexes use logical rowids to locate table rows.
- Bitmap Indexes on Index-Organized Tables
  A secondary index on an index-organized table can be a **bitmap index**. A bitmap index stores a bitmap for each index key.

> **See Also:**
>
> - "Rowid Data Types" to learn more about the use of rowids, and the `ROWID` pseudocolumn
> - "Chained and Migrated Rows "
>
>   to learn why rows migrate, and why migration increases the number of I/Os
> - *Oracle Database Administrator's Guide* to learn how to create secondary indexes on an index-organized table
> - *Oracle Database VLDB and Partitioning Guide* to learn about creating secondary indexes on indexed-organized table partitions

## Logical Rowids and Physical Guesses

Secondary indexes use logical rowids to locate table rows.

A logical rowid includes a **physical guess**, which is the physical rowid of the index entry when it was first made. Oracle Database can use physical guesses to probe directly into the leaf block of the index-organized table, bypassing the primary key search. When the physical location of a row changes, the logical rowid remains valid even if it contains a physical guess that is stale.

For a heap-organized table, access by a secondary index involves a scan of the secondary index and an additional I/O to fetch the data block containing the row. For index-organized tables, access by a secondary index varies, depending on the use and accuracy of physical guesses:

- Without physical guesses, access involves two index scans: a scan of the secondary index followed by a scan of the primary key index.
- With physical guesses, access depends on their accuracy:
  - With accurate physical guesses, access involves a secondary index scan and an additional I/O to fetch the data block containing the row.
  - With inaccurate physical guesses, access involves a secondary index scan and an I/O to fetch the wrong data block (as indicated by the guess), followed by an index unique scan of the index organized table by primary key value.

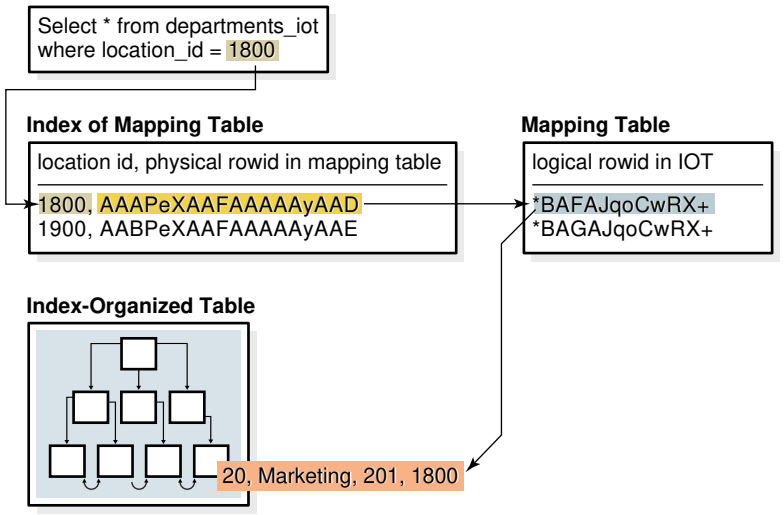
## Bitmap Indexes on Index-Organized Tables

A secondary index on an index-organized table can be a **bitmap index**. A bitmap index stores a bitmap for each index key.

When bitmap indexes exist on an index-organized table, all the bitmap indexes use a heap-organized mapping table. The mapping table stores the logical rowids of the index-organized table. Each mapping table row stores one logical rowid for the corresponding index-organized table row.

The database accesses a bitmap index using a search key. If the database finds the key, then the bitmap entry is converted to a physical rowid. With heap-organized tables, the database uses the physical rowid to access the base table. With index-organized tables, the database uses the physical rowid to access the mapping table, which in turn yields a logical rowid that the database uses to access the index-organized table. The following figure illustrates index access for a query of the `departments_iot` table.

**Figure 5-4 Bitmap Index on Index-Organized Table**




> **Note:**
>
> Movement of rows in an index-organized table does not leave the bitmap indexes built on that index-organized table unusable.

> **See Also:**
>
> "Rowids of Row Pieces" to learn about the differences between physical and logical rowids

6

# Partitions, Views, and Other Schema Objects

Although tables and indexes are the most important and commonly used schema objects, the database supports many other types of schema objects, the most common of which are discussed in this chapter.

This chapter contains the following sections:


- Overview of Partitions
- Overview of Sharded Tables
- Overview of Views
- Overview of Materialized Views
- Overview of Sequences
- Overview of Dimensions
- Overview of Synonyms


- Overview of Partitions
  In an Oracle database, **partitioning** enables you to decompose very large tables and indexes into smaller and more manageable pieces called **partitions**. Each partition is an independent object with its own name and optionally its own storage characteristics.
- Overview of Sharded Tables
  In an Oracle database, sharding enables you to break up a large table into more manageable pieces called shards that can be stored in multiple databases.
- Overview of Views
  A **view** is a logical representation of one or more tables. In essence, a view is a stored query.
- Overview of Materialized Views
  A **materialized view** is a query result that has been stored or "materialized" in advance as a schema object. The `FROM` clause of the query can name tables, views, or materialized views.
- Overview of Sequences
  A **sequence** is a schema object from which multiple users can generate unique integers. A sequence generator provides a highly scalable and well-performing method to generate surrogate keys for a number data type.
- Overview of Dimensions
  A typical data warehouse has two important components: dimensions and facts.
- Overview of Synonyms
  A **synonym** is an alias for a schema object. For example, you can create a synonym for a table or view, sequence, PL/SQL program unit, user-defined object type, or another synonym. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

# Overview of Partitions

In an Oracle database, **partitioning** enables you to decompose very large tables and indexes into smaller and more manageable pieces called **partitions**. Each partition is an independent object with its own name and optionally its own storage characteristics.

For an analogy that illustrates partitioning, suppose an HR manager has one big box that contains employee folders. Each folder lists the employee hire date. Queries are often made for employees hired in a particular month. One approach to satisfying such requests is to create an index on employee hire date that specifies the locations of the folders scattered throughout the box. In contrast, a partitioning strategy uses many smaller boxes, with each box containing folders for employees hired in a given month.

Using smaller boxes has several advantages. When asked to retrieve the folders for employees hired in June, the HR manager can retrieve the June box. Furthermore, if any small box is temporarily damaged, the other small boxes remain available. Moving offices also becomes easier because instead of moving a single heavy box, the manager can move several small boxes.

From the perspective of an application, only one schema object exists. SQL statements require no modification to access partitioned tables. Partitioning is useful for many different types of database applications, particularly those that manage large volumes of data. Benefits include:

- Increased availability

  The unavailability of a partition does not entail the unavailability of the object. The query optimizer automatically removes unreferenced partitions from the query plan so queries are not affected when the partitions are unavailable.

- Easier administration of schema objects

  A partitioned object has pieces that can be managed either collectively or individually. DDL statements can manipulate partitions rather than entire tables or indexes. Thus, you can break up resource-intensive tasks such as rebuilding an index or table. For example, you can move one table partition at a time. If a problem occurs, then only the partition move must be redone, not the table move. Also, dropping a partition avoids executing numerous `DELETE` statements.

- Reduced contention for shared resources in OLTP systems

  In some OLTP systems, partitions can decrease contention for a shared resource. For example, DML is distributed over many segments rather than one segment.

- Enhanced query performance in data warehouses

  In a data warehouse, partitioning can speed processing of ad hoc queries. For example, a sales table containing a million rows can be partitioned by quarter.

- Partition Characteristics
  Each partition of a table or index must have the same logical attributes, such as column names, data types, and constraints.
- Partitioned Tables
  A **partitioned table** consists of one or more partitions, which are managed individually and can operate independently of the other partitions.
- Partitioned Indexes
  A **partitioned index** is an index that, like a partitioned table, has been divided into smaller and more manageable pieces.

- Partial Indexes for Partitioned Tables
  A **partial index** is an index that is correlated with the indexing properties of an associated partitioned table.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for an introduction to partitioning

# Partition Characteristics

Each partition of a table or index must have the same logical attributes, such as column names, data types, and constraints.

For example, all partitions in a table share the same column and constraint definitions. However, each partition can have separate physical attributes, such as the tablespace to which it belongs.

- Partition Key
  The **partition key** is a set of one or more columns that determines the partition in which each row in a partitioned table should go. Each row is unambiguously assigned to a single partition.
- Partitioning Strategies
  Oracle Partitioning offers several partitioning strategies that control how the database places data into partitions. The basic strategies are range, list, and hash partitioning.

## Partition Key

The **partition key** is a set of one or more columns that determines the partition in which each row in a partitioned table should go. Each row is unambiguously assigned to a single partition.

In the `sales` table, you could specify the `time_id` column as the key of a range partition. The database assigns rows to partitions based on whether the date in this column falls in a specified range. Oracle Database automatically directs insert, update, and delete operations to the appropriate partition by using the partition key.

## Partitioning Strategies

Oracle Partitioning offers several partitioning strategies that control how the database places data into partitions. The basic strategies are range, list, and hash partitioning.

A single-level partitioning uses only one method of data distribution, for example, only list partitioning or only range partitioning. In composite partitioning, a table is partitioned by one data distribution method and then each partition is further divided into subpartitions using a second data distribution method. For example, you could use a list partition for `channel_id` and a range subpartition for `time_id`.

**Example 6-1 Sample Row Set for Partitioned Table**

This partitioning example assumes that you want to populate a partitioned table `sales` with the following rows:

```
  PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
```

```
       116      11393 05-JUN-99          2         999             1        12.18
        40     100530 30-NOV-98          9          33             1        44.99
       118        133 06-JUN-01          2         999             1        17.12
       133       9450 01-DEC-00          2         999             1        31.28
        36       4523 27-JAN-99          3         999             1        53.89
       125       9417 04-FEB-98          3         999             1        16.86
        30        170 23-FEB-01          2         999             1          8.8
        24      11899 26-JUN-99          4         999             1        43.04
        35       2606 17-FEB-00          3         999             1        54.94
        45       9491 28-AUG-98          4         350             1
47.45
```

- Range Partitioning
  In **range partitioning**, the database maps rows to partitions based on ranges of values of the partitioning key. Range partitioning is the most common type of partitioning and is often used with dates.
- Interval Partitioning
  **Interval partitioning** is an extension of range partitioning.
- List Partitioning
  In **list partitioning**, the database uses a list of discrete values as the partition key for each partition. The partitioning key consists of one or more columns.
- Hash Partitioning
  In **hash partitioning**, the database maps rows to partitions based on a hashing algorithm that the database applies to the user-specified partitioning key.
- Reference Partitioning
  In **reference partitioning**, the partitioning strategy of a child table is solely defined through the foreign key relationship with a parent table. For every partition in the parent table, exactly one corresponding partition exists in the child table. The parent table stores the parent records in a specific partition, and the child table stores the child records in the corresponding partition.
- Composite Partitioning
  In **composite partitioning**, a table is partitioned by one data distribution method and then each partition is further subdivided into subpartitions using a second data distribution method. Thus, composite partitioning combines the basic data distribution methods. All subpartitions for a given partition represent a logical subset of the data.

## Range Partitioning

In **range partitioning**, the database maps rows to partitions based on ranges of values of the partitioning key. Range partitioning is the most common type of partitioning and is often used with dates.

Suppose that you create `time_range_sales` as a partitioned table using the following SQL statement, with the `time_id` column as the partition key:

```
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_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
  PARTITION SALES_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
  PARTITION SALES_2000 VALUES LESS THAN (TO_DATE('01-JAN-2001','DD-MON-YYYY')),
  PARTITION SALES_2001 VALUES LESS THAN (MAXVALUE)
 );
```

Afterward, you load `time_range_sales` with the rows from Example 6-1. The code shows the row distributions in the four partitions. The database chooses the partition for each row based on the `time_id` value according to the rules specified in the `PARTITION BY RANGE` clause. The range partition key value determines the non-inclusive high bound for a specified partition.

## Interval Partitioning

**Interval partitioning** is an extension of range partitioning.

If you insert data that exceeds existing range partitions, then Oracle Database automatically creates partitions of a specified interval. For example, you could create a sales history table that stores data for each month in a separate partition.

Interval partitions enable you to avoid creating range partitions explicitly. You can use interval partitioning for almost every table that is range partitioned and uses fixed intervals for new partitions. Unless you create range partitions with different intervals, or unless you always set specific partition attributes, consider using interval partitions.

When partitioning by interval, you must specify at least one range partition. The range partitioning key value determines the high value of the range partitions, which is called the transition point. The database automatically creates interval partitions for data with values that are beyond the transition point. The lower boundary of every interval partition is the inclusive upper boundary of the previous range or interval partition. Thus, in Example 6-2, value `01-JAN-2011` is in partition `p2`.

The database creates interval partitions for data beyond the transition point. An interval partition extends range partitioning by instructing the database to create partitions of the specified range or interval. The database automatically creates the partitions when data inserted into the table exceeds all existing range partitions. In Example 6-2, the `p3` partition contains rows with partitioning key `time_id` values greater than or equal to `01-JAN-2013`.

**Example 6-2 Interval Partitioning**

Assume that you create a sales table with four partitions of varying widths. You specify that above the transition point of January 1, 2013, the database should create partitions in one month intervals. The high bound of partition `p3` represents the transition point. Partition `p3` and all partitions below it are in the range section, whereas all partitions above it fall into the interval section.

```
CREATE TABLE interval_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)
  INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))
    ( PARTITION p0 VALUES LESS THAN (TO_DATE('1-1-2010', 'DD-MM-YYYY'))
    , PARTITION p1 VALUES LESS THAN (TO_DATE('1-1-2011', 'DD-MM-YYYY'))
```

```
    , PARTITION p2 VALUES LESS THAN (TO_DATE('1-7-2012', 'DD-MM-YYYY'))
    , PARTITION p3 VALUES LESS THAN (TO_DATE('1-1-2013', 'DD-MM-YYYY')) );
```

You insert a sale made on date October 10, 2014:

```
SQL> INSERT INTO interval_sales VALUES (39,7602,'10-OCT-14',9,null,1,11.79);

1 row created.
```

A query of `USER_TAB_PARTITIONS` shows that the database created a new partition for the October 10 sale because the sale date was later than the transition point:

```
SQL> COL PNAME FORMAT a9
SQL> COL HIGH_VALUE FORMAT a40
SQL> SELECT PARTITION_NAME AS PNAME, HIGH_VALUE
  2  FROM USER_TAB_PARTITIONS WHERE TABLE_NAME = 'INTERVAL_SALES';

PNAME     HIGH_VALUE
--------- ----------------------------------------
P0        TO_DATE(' 2007-01-01 00:00:00', 'SYYYY-M
          M-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
P1        TO_DATE(' 2008-01-01 00:00:00', 'SYYYY-M
          M-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
P2        TO_DATE(' 2009-07-01 00:00:00', 'SYYYY-M
          M-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
P3        TO_DATE(' 2010-01-01 00:00:00', 'SYYYY-M
          M-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
SYS_P1598 TO_DATE(' 2014-11-01 00:00:00', 'SYYYY-M
          M-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
```

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* to learn more about interval partitions

## List Partitioning

In **list partitioning**, the database uses a list of discrete values as the partition key for each partition. The partitioning key consists of one or more columns.

You can use list partitioning to control how individual rows map to specific partitions. By using lists, you can group and organize related sets of data when the key used to identify them is not conveniently ordered.

**Example 6-3 List Partitioning**

Assume that you create `list_sales` as a list-partitioned table using the following statement, where the `channel_id` column is the partition key:

```
CREATE TABLE list_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 LIST (channel_id)
 ( PARTITION even_channels VALUES ('2','4'),
   PARTITION odd_channels VALUES ('3','9')
 );
```

Afterward, you load the table with the rows from Example 6-1. The code shows the row distribution in the two partitions. The database chooses the partition for each row based on the `channel_id` value according to the rules specified in the `PARTITION BY LIST` clause. Rows with a `channel_id` value of 2 or 4 are stored in the `EVEN_CHANNELS` partitions, while rows with a `channel_id` value of 3 or 9 are stored in the `ODD_CHANNELS` partition.

## Hash Partitioning

In **hash partitioning**, the database maps rows to partitions based on a hashing algorithm that the database applies to the user-specified partitioning key.

The destination of a row is determined by the internal hash function applied to the row by the database. When the number of partitions is a power of 2, the hashing algorithm creates a roughly even distribution of rows across all partitions.

Hash partitioning is useful for dividing large tables to increase manageability. Instead of one large table to manage, you have several smaller pieces. The loss of a single hash partition does not affect the remaining partitions and can be recovered independently. Hash partitioning is also useful in OLTP systems with high update contention. For example, a segment is divided into several pieces, each of which is updated, instead of a single segment that experiences contention.

Assume that you create the partitioned `hash_sales` table using the following statement, with the `prod_id` column as the partition key:

```
CREATE TABLE hash_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 HASH (prod_id)
PARTITIONS 2;
```

Afterward, you load the table with the rows from Example 6-1. The code shows a possible row distribution in the two partitions. The names of these partitions are system-generated.

As you insert rows, the database attempts to randomly and evenly distribute them across partitions. You cannot specify the partition into which a row is placed. The database applies the hash function, whose outcome determines which partition contains the row.

> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide* to learn how to create partitions
> - *Oracle Database SQL Language Reference* for `CREATE TABLE ... PARTITION BY` examples

## Reference Partitioning

In **reference partitioning**, the partitioning strategy of a child table is solely defined through the foreign key relationship with a parent table. For every partition in the parent table, exactly one corresponding partition exists in the child table. The parent table stores the parent records in a specific partition, and the child table stores the child records in the corresponding partition.

For example, an `orders` table is the parent of the `line_items` table, with a primary key and foreign key defined on `order_id`. The tables are partitioned by reference. For example, if the database stores order 233 in partition `Q3_2015` of `orders`, then the database stores all line items for order 233 in partition `Q3_2015` of `line_items`. If partition `Q4_2015` is added to `orders`, then the database automatically adds `Q4_2015` to `line_items`.

The advantages of reference partitioning are:

- By using the same partitioning strategy for both the parent and child tables, you avoid duplicating all partitioning key columns. This strategy reduces the manual overhead of denormalization, and saves space.
- Maintenance operations on a parent table occur on the child table automatically. For example, when you add a partition to the master table, the database automatically propagates this addition to its descendents.
- The database automatically uses partition-wise joins of the partitions in the parent and child table, improving performance.

You can use reference partitioning with all basic partitioning strategies, including interval partitioning. You can also create reference partitioned tables as composite partitioned tables.

**Example 6-4 Creating Reference-Partitioned Tables**

This example creates a parent table orders which is range-partitioned on `order_date`. The reference-partitioned child table `order_items` is created with four partitions, `Q1_2015`, `Q2_2015`, `Q3_2015`, and `Q4_2015`, where each partition contains the `order_items` rows corresponding to orders in the respective parent partition.

```
CREATE TABLE orders
    ( order_id           NUMBER(12),
      order_date         DATE,
      order_mode         VARCHAR2(8),
      customer_id        NUMBER(6),
      order_status       NUMBER(2),
      order_total        NUMBER(8,2),
      sales_rep_id       NUMBER(6),
      promotion_id       NUMBER(6),
      CONSTRAINT orders_pk PRIMARY KEY(order_id)
    )
  PARTITION BY RANGE(order_date)
    ( PARTITION Q1_2015 VALUES LESS THAN (TO_DATE('01-APR-2015','DD-MON-
```

```
YYYY')),
      PARTITION Q2_2015 VALUES LESS THAN (TO_DATE('01-JUL-2015','DD-MON-
YYYY')),
      PARTITION Q3_2015 VALUES LESS THAN (TO_DATE('01-OCT-2015','DD-MON-
YYYY')),
      PARTITION Q4_2015 VALUES LESS THAN (TO_DATE('01-JAN-2006','DD-MON-
YYYY'))
    );

CREATE TABLE order_items
    ( order_id           NUMBER(12) NOT NULL,
      line_item_id       NUMBER(3)  NOT NULL,
      product_id         NUMBER(6)  NOT NULL,
      unit_price         NUMBER(8,2),
      quantity           NUMBER(8),
      CONSTRAINT order_items_fk
      FOREIGN KEY(order_id) REFERENCES orders(order_id)
    )
    PARTITION BY REFERENCE(order_items_fk);
```

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for an overview of reference partitioning

## Composite Partitioning

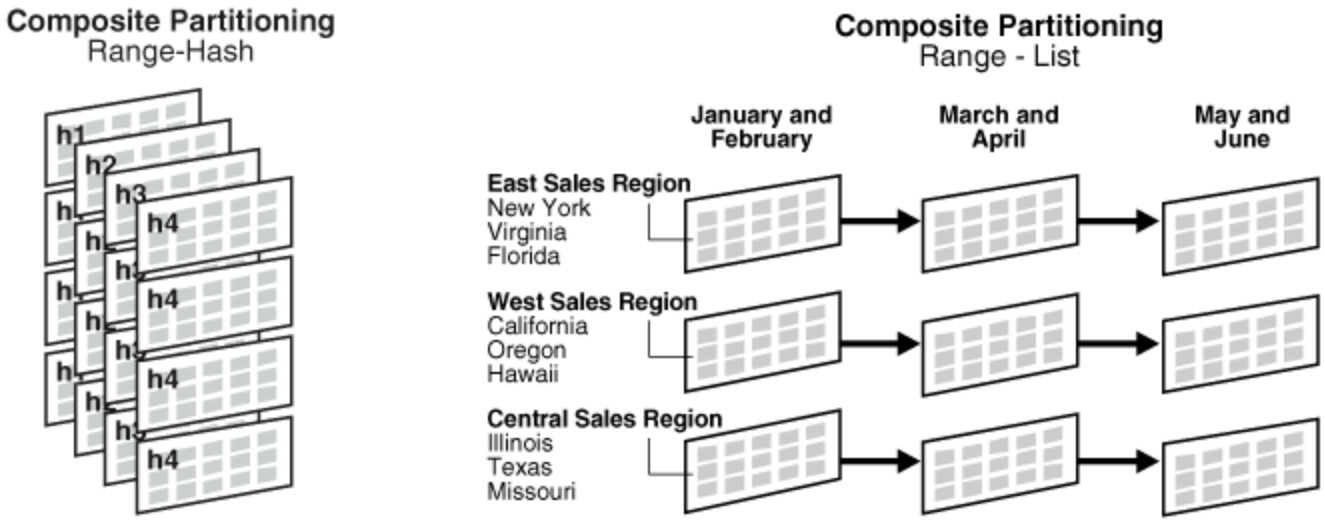
In **composite partitioning**, a table is partitioned by one data distribution method and then each partition is further subdivided into subpartitions using a second data distribution method. Thus, composite partitioning combines the basic data distribution methods. All subpartitions for a given partition represent a logical subset of the data.

Composite partitioning provides several advantages:

- Depending on the SQL statement, partition pruning on one or two dimensions may improve performance.
- Queries may be able to use full or partial partition-wise joins on either dimension.
- You can perform parallel backup and recovery of a single table.
- The number of partitions is greater than in single-level partitioning, which may be beneficial for parallel execution.
- You can implement a rolling window to support historical data and still partition on another dimension if many statements can benefit from partition pruning or partition-wise joins.
- You can store data differently based on identification by a partitioning key. For example, you may decide to store data for a specific product type in a read-only, compressed format, and keep other product type data uncompressed.

Range, list, and hash partitioning are eligible as subpartitioning strategies for composite partitioned tables. The following figure offers a graphical view of range-hash and range-list composite partitioning.

**Figure 6-1 Composite Range-List Partitioning**




The database stores every subpartition in a composite partitioned table as a separate segment. Thus, subpartition properties may differ from the properties of the table or from the partition to which the subpartitions belong.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* to learn more about composite partitioning

# Partitioned Tables

A **partitioned table** consists of one or more partitions, which are managed individually and can operate independently of the other partitions.

A table is either partitioned or nonpartitioned. Even if a partitioned table consists of only one partition, this table is different from a nonpartitioned table, which cannot have partitions added to it.

- Segments for Partitioned Tables
  A partitioned table is made up of one or more table partition segments.
- Compression for Partitioned Tables
  Some or all partitions of a heap-organized table can be stored in a compressed format.

> **See Also:**
>
> "Partition Characteristics" for examples of partitioned tables
>
> "Overview of Index-Organized Tables" to learn about the purpose and characteristics of Index-Organized Tables, which can also benefit from partitioning that provides improved manageability, availability, and performance.

## Segments for Partitioned Tables

A partitioned table is made up of one or more table partition segments.

If you create a partitioned table named `hash_products`, then no table segment is allocated for this table. Instead, the database stores data for each table partition in its own partition segment. Each table partition segment contains a portion of the table data.

When an external table is partitioned, all partitions reside outside the database. In a hybrid partitioned table, some partitions are stored in segments, whereas others are stored externally. For example, some partitions of the `sales` table might be stored in data files and others in spreadsheets.

> **See Also:**
>
> - "Overview of External Tables"
> - "Overview of Segments"
>
>  to learn about the relationship between objects and segments

## Compression for Partitioned Tables

Some or all partitions of a heap-organized table can be stored in a compressed format.

Compression saves space and can speed query execution. For this reason, compression can be useful in environments such as data warehouses, where the amount of insert and update operations is small, and in OLTP environments.

You can declare the attributes for table compression for a tablespace, table, or table partition. If declared at the tablespace level, then tables created in the tablespace are compressed by default. You can alter the compression attribute for a table, in which case the change only applies to new data going into that table. Consequently, a single table or partition may contain compressed and uncompressed blocks, which guarantees that data size will not increase because of compression. If compression could increase the size of a block, then the database does not apply it to the block.

> **See Also:**
>
> - "Table Compression" to learn about types of table compression, including basic, advanced row, and Hybrid Columnar Compression
> - *Oracle Database Data Warehousing Guide* to learn about table compression in a data warehouse

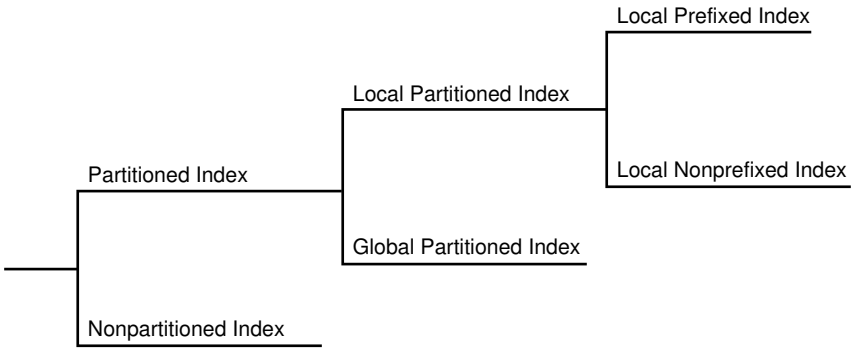
## Partitioned Indexes

A **partitioned index** is an index that, like a partitioned table, has been divided into smaller and more manageable pieces.

Global indexes are partitioned independently of the table on which they are created, whereas local indexes are automatically linked to the partitioning method for a table. Like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability.

The following graphic shows index partitioning options.

**Figure 6-2 Index Partitioning Options**




- Local Partitioned Indexes
  In a **local partitioned index**, the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as its table.
- Global Partitioned Indexes
  A **global partitioned index** is a B-tree index that is partitioned independently of the underlying table on which it is created. A single index partition can point to any or all table partitions, whereas in a locally partitioned index, a one-to-one parity exists between index partitions and table partitions.

> **See Also:**
>
> - "Introduction to Indexes" to learn about the difference between unique and nonunique indexes, and the different index types
> - *Oracle Database VLDB and Partitioning Guide* for more information about partitioned indexes and how to decide which type to use

## Local Partitioned Indexes

In a **local partitioned index**, the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as its table.

Each index partition is associated with exactly one partition of the underlying table, so that all keys in an index partition refer only to rows stored in a single table partition. In this way, the database automatically synchronizes index partitions with their associated table partitions, making each table-index pair independent.

Local partitioned indexes are common in data warehousing environments. Local indexes offer the following advantages:

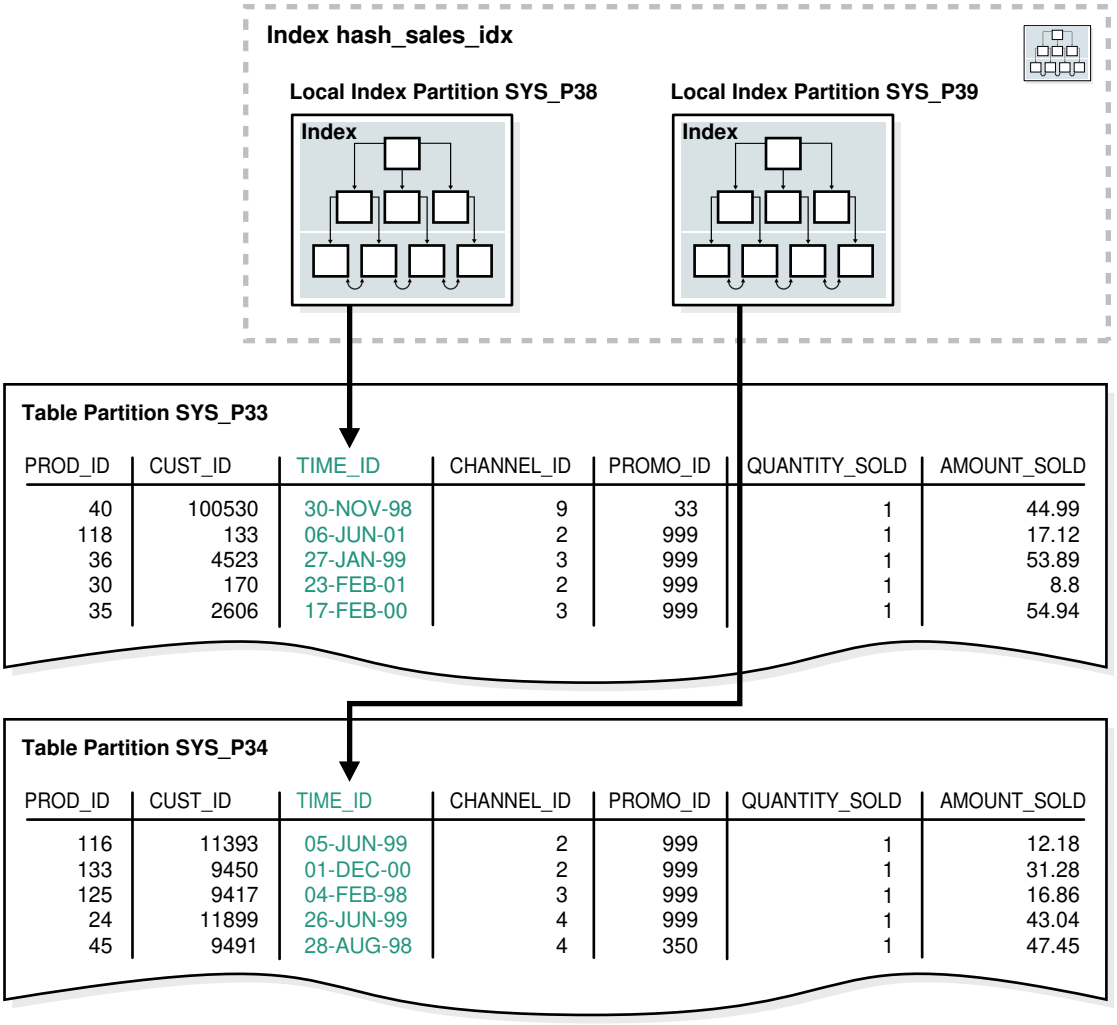
- Availability is increased because actions that make data invalid or unavailable in a partition affect this partition only.
- Partition maintenance is simplified. When moving a table partition, or when data ages out of a partition, only the associated local index partition must be rebuilt or maintained. In a global index, all index partitions must be rebuilt or maintained.
- If point-in-time recovery of a partition occurs, then the indexes can be recovered to the recovery time (see *Oracle Database Backup and Recovery User's Guide*). The entire index does not need to be rebuilt.

The example in Hash Partitioning shows the creation statement for the partitioned `hash_sales` table, using the `prod_id` column as partition key. The following example creates a local partitioned index on the `time_id` column of the `hash_sales` table:

```
CREATE INDEX hash_sales_idx ON hash_sales(time_id) LOCAL;
```

In Figure 6-3, the `hash_products` table has two partitions, so `hash_sales_idx` has two partitions. Each index partition is associated with a different table partition. Index partition `SYS_P38` indexes rows in table partition `SYS_P33`, whereas index partition `SYS_P39` indexes rows in table partition `SYS_P34`.

**Figure 6-3 Local Index Partitions**




You cannot explicitly add a partition to a local index. Instead, new partitions are added to local indexes only when you add a partition to the underlying table. Likewise, you cannot explicitly drop a partition from a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.

Like other indexes, you can create a bitmap index on partitioned tables. The only restriction is that bitmap indexes must be local to the partitioned table—they cannot be global indexes. Global bitmap indexes are supported only on nonpartitioned tables.

- Local Prefixed and Nonprefixed Indexes
  Local partitioned indexes are either prefixed or nonprefixed.
- Local Partitioned Index Storage
  Like a table partition, a local index partition is stored in its own segment. Each segment contains a portion of the total index data. Thus, a local index made up of four partitions is not stored in a single index segment, but in four separate segments.

## Local Prefixed and Nonprefixed Indexes

Local partitioned indexes are either prefixed or nonprefixed.

The index subtypes are defined as follows:

- Local prefixed indexes

  In this case, the partition keys are on the leading edge of the index definition. In the `time_range_sales` example in Range Partitioning, the table is partitioned by range on `time_id`. A local prefixed index on this table would have `time_id` as the first column in its list.

- Local nonprefixed indexes

  In this case, the partition keys are not on the leading edge of the indexed column list and need not be in the list at all. In the `hash_sales_idx` example in Local Partitioned Indexes, the index is local nonprefixed because the partition key `product_id` is not on the leading edge.

Both types of indexes can take advantage of partition elimination (also called *partition pruning*), which occurs when the optimizer speeds data access by excluding partitions from consideration. Whether a query can eliminate partitions depends on the query predicate. A query that uses a local prefixed index always allows for index partition elimination, whereas a query that uses a local nonprefixed index might not.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* to learn how to use prefixed and nonprefixed indexes

### Local Partitioned Index Storage

Like a table partition, a local index partition is stored in its own segment. Each segment contains a portion of the total index data. Thus, a local index made up of four partitions is not stored in a single index segment, but in four separate segments.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `CREATE INDEX ... LOCAL` examples

## Global Partitioned Indexes

A **global partitioned index** is a B-tree index that is partitioned independently of the underlying table on which it is created. A single index partition can point to any or all table partitions, whereas in a locally partitioned index, a one-to-one parity exists between index partitions and table partitions.

In general, global indexes are useful for OLTP applications, where rapid access, data integrity, and availability are important. In an OLTP system, a table may be partitioned by one key, for example, the `employees.department_id` column, but an application may need to access the data with many different keys, for example, by `employee_id` or `job_id`. Global indexes can be useful in this scenario.

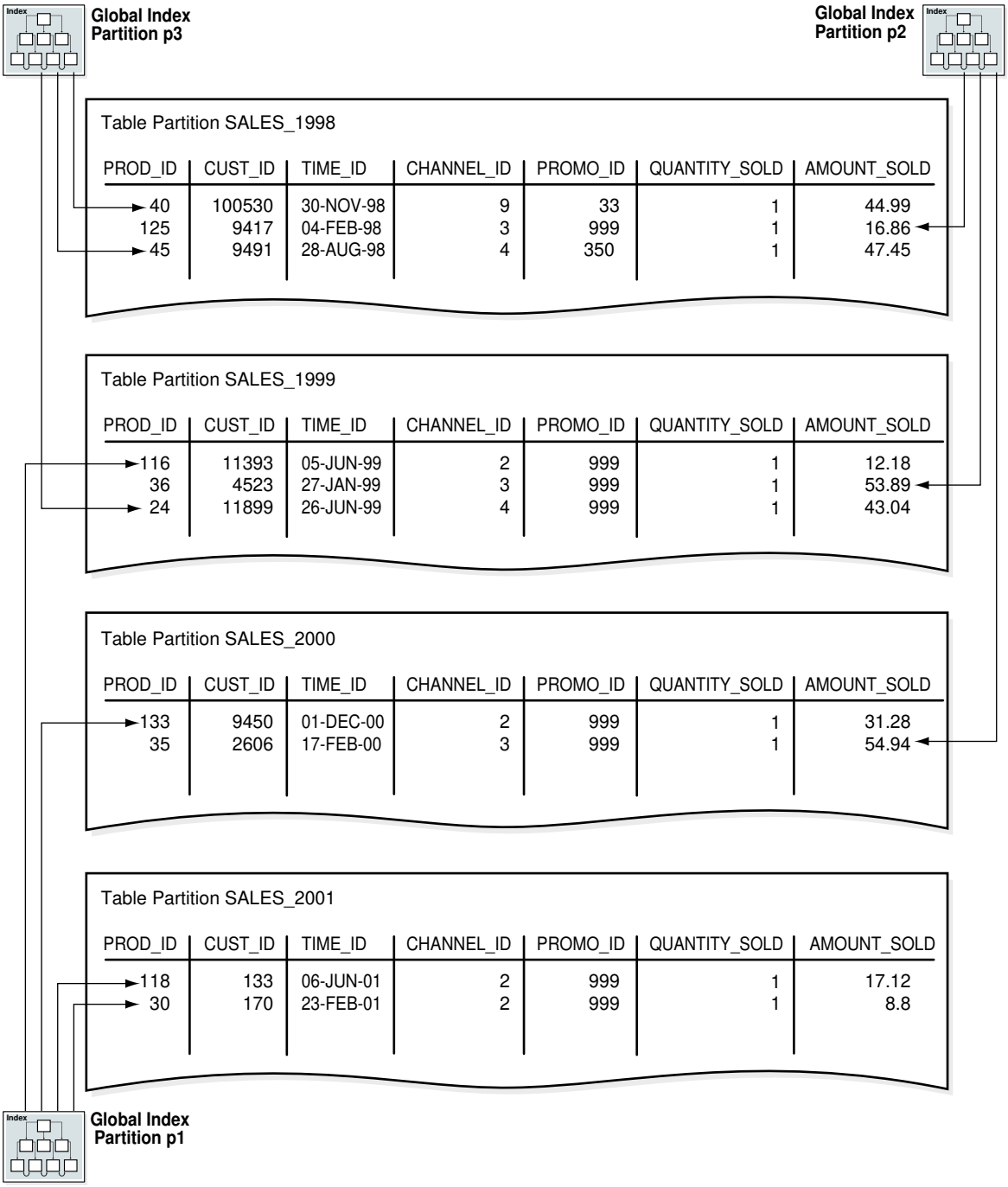
As an illustration, suppose that you create a global partitioned index on the `time_range_sales` table from "Range Partitioning". In this table, rows for sales from 1998 are stored in one

partition, rows for sales from 1999 are in another, and so on. The following example creates a global index partitioned by range on the `channel_id` column:

```
CREATE INDEX time_channel_sales_idx ON time_range_sales (channel_id)
   GLOBAL PARTITION BY RANGE (channel_id)
      (PARTITION p1 VALUES LESS THAN (3),
       PARTITION p2 VALUES LESS THAN (4),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

As shown in Figure 6-4, a global index partition can contain entries that point to multiple table partitions. Index partition `p1` points to the rows with a `channel_id` of 2, index partition `p2` points to the rows with a `channel_id` of 3, and index partition `p3` points to the rows with a `channel_id` of 4 or 9.

**Figure 6-4 Global Partitioned Index**



> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide* to learn how to manage global partitioned indexes
> - *Oracle Database SQL Language Reference* to learn about the `GLOBAL PARTITION` clause of `CREATE INDEX`

## Partial Indexes for Partitioned Tables

A **partial index** is an index that is correlated with the indexing properties of an associated partitioned table.

The correlation enables you to specify which table partitions are indexed. Partial indexes provide the following advantages:

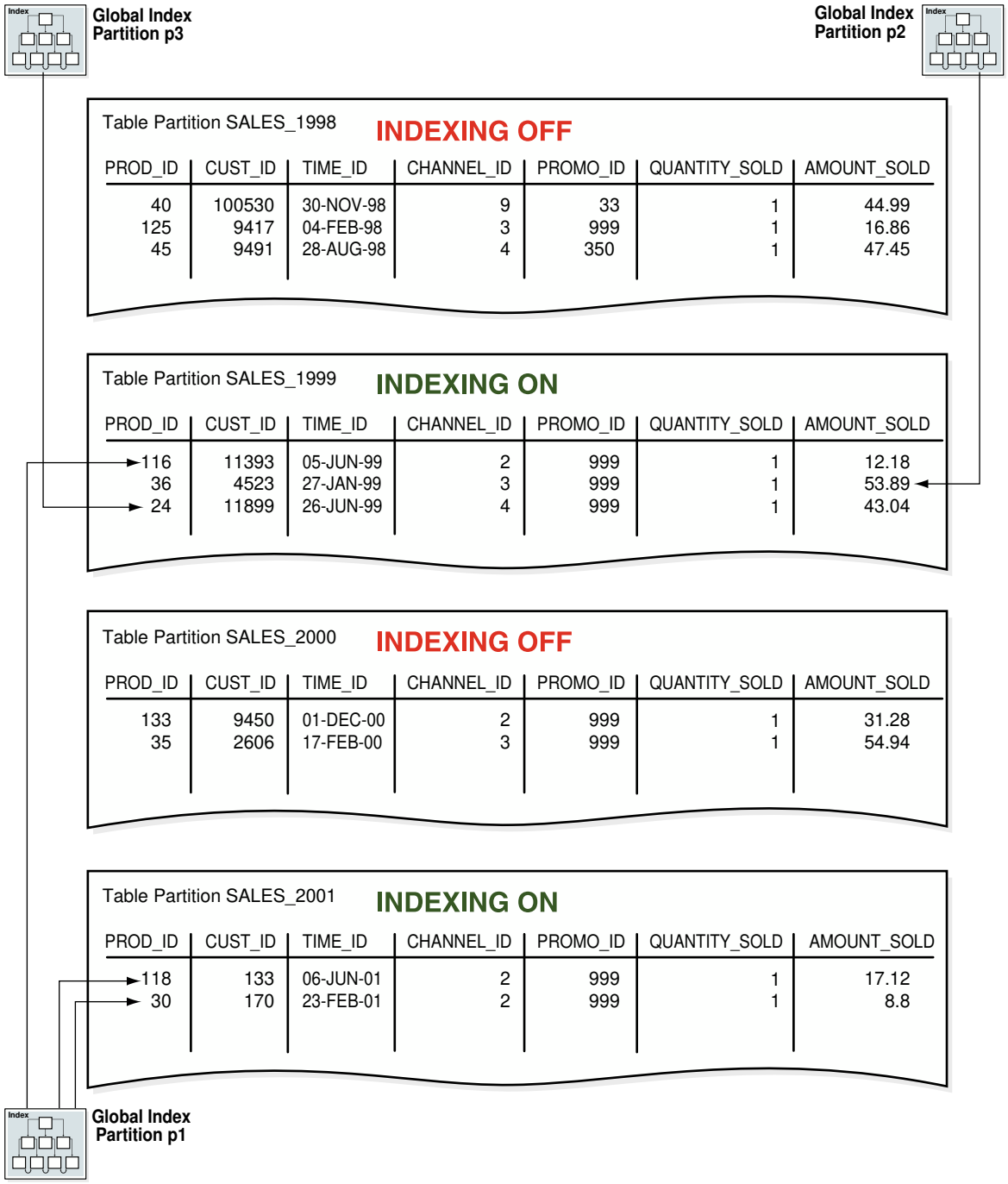
- Table partitions that are not indexed avoid consuming unnecessary index storage space.
- Performance of loads and queries can improve.

  Before Oracle Database 12c, an exchange partition operation required a physical update of an associated global index to retain it as usable. Starting with Oracle Database 12c, if the partitions involved in a partition maintenance operation are not part of a partial global index, then the index remains usable without requiring any global index maintenance.
- If you index only some table partitions at index creation, and if you later index other partitions, then you can reduce the sort space required by index creation.

You can turn indexing on or off for the individual partitions of a table. A partial local index does not have usable index partitions for all table partitions that have indexing turned off. A global index, whether partitioned or not, excludes the data from all partitions that have indexing turned off. The database does not support partial indexes for indexes that enforce unique constraints.

Figure 6-5 shows the same global index as in Figure 6-4, except that the global index is partial. Table partitions `SALES_1998` and `SALES_2000` have the indexing property set to `OFF`, so the partial global index does not index them.

**Figure 6-5 Partial Global Partitioned Index**




# Overview of Sharded Tables

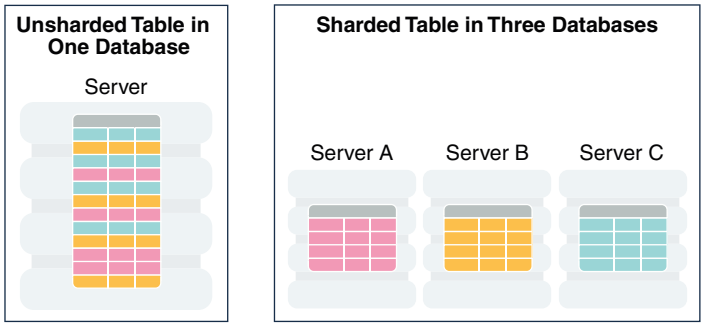
In an Oracle database, sharding enables you to break up a large table into more manageable pieces called shards that can be stored in multiple databases.

Each database is hosted on a dedicated server with its own local resources - CPU, memory, flash, or disk. Each database in such configuration is called a **shard**. All of the shards together make up a single logical database, which is referred to as a **sharded database**.

Horizontal partitioning involves splitting a database table across shards so that each shard contains the table with the same columns but a different subset of rows. A table split up in this manner is also known as a **sharded table**.

The following figure shows a table horizontally partitioned across three shards.

**Figure 6-6 Horizontal Partitioning of a Table Across Shards**




Sharding is based on shared-nothing hardware infrastructure and it eliminates single points of failure because shards do not share physical resources such as CPU, memory, or storage devices. Shards are also loosely coupled in terms of software; they do not run clusterware.

Shards are typically hosted on dedicated servers. These servers can be commodity hardware or engineered systems. The shards can run on single instance or Oracle RAC databases. They can be placed on-premises, in a cloud, or in a hybrid on-premises and cloud configuration.

From the perspective of a database administrator, an SDB consists of multiple databases that can be managed either collectively or individually. However, from the perspective of the application, an SDB looks like a single database: the number of shards and distribution of data across those shards are completely transparent to database applications.

Sharding is intended for custom OLTP applications that are suitable for a sharded database architecture. Applications that use sharding must have a well-defined data model and data distribution strategy (consistent hash, range, list, or composite) that primarily accesses data using a sharding key. Examples of a sharding key include `customer_id`, `account_no`, or `country_id`.

- Sharded Tables
  A sharded table is a table that is partitioned into smaller and more manageable pieces among multiple databases, called shards.

> **See Also:**
>
> *Using Oracle Sharding*

## Sharded Tables

A sharded table is a table that is partitioned into smaller and more manageable pieces among multiple databases, called shards.

Partitions are distributed across shards at the tablespace level, based on a sharding key. Examples of keys include customer ID, account number, and country ID. The following data types are supported for the sharding key.

- NUMBER
- INTEGER
- SMALLINT
- RAW
- (N)VARCHAR
- (N)VARCHAR2
- (N)CHAR
- DATE
- TIMESTAMP

Each partition of a sharded table resides in a separate tablespace, and each tablespace is associated with a specific shard. Depending on the sharding method, the association can be established automatically or defined by the administrator.

Even though the partitions of a sharded table reside in multiple shards, to the application, the table looks and behaves exactly the same as a partitioned table in a single database. SQL statements issued by an application never have to refer to shards or depend on the number of shards and their configuration.

The familiar SQL syntax for table partitioning specifies how rows should be partitioned across shards. For example, the following SQL statement creates a sharded table, horizontally partitioning the table across shards based on the sharding key `cust_id`.

```
CREATE SHARDED TABLE customers
( cust_id     NUMBER NOT NULL
, name        VARCHAR2(50)
, address     VARCHAR2(250)
, region      VARCHAR2(20)
, class       VARCHAR2(3)
, signup      DATE
CONSTRAINT cust_pk PRIMARY KEY(cust_id)
)
PARTITION BY CONSISTENT HASH (cust_id)
PARTITIONS AUTO
TABLESPACE SET ts1
;
```

The sharded table is partitioned by consistent hash, a special type of hash partitioning commonly used in scalable distributed systems. This technique automatically spreads tablespaces across shards to provide an even distribution of data and workload.

> **Note:**
>
> Global indexes on sharded tables are not supported, but local indexes are supported.

**Related Topics**

- Sharding Architecture
  Oracle Sharding is a database scaling technique based on horizontal partitioning of data across multiple PDBs. Applications perceive the pool of PDBs as a single logical database.
- *Using Oracle Sharding*

# Overview of Views

A **view** is a logical representation of one or more tables. In essence, a view is a stored query.

A view derives its data from the tables on which it is based, called *base tables*. Base tables can be tables or other views. All operations performed on a view actually affect the base tables. You can use views in most places where tables are used.

> **Note:**
>
> Materialized views use a different data structure from standard views.

Views enable you to tailor the presentation of data to different types of users. Views are often used to:

- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table

  For example, Figure 6-7 shows how the `staff` view does not show the `salary` or `commission_pct` columns of the base table `employees`.
- Hide data complexity

  For example, a single view can be defined with a join, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables. A query might also perform extensive calculations with table information. Thus, users can query a view without knowing how to perform a join or calculations.
- Present the data in a different perspective from that of the base table

  For example, the columns of a view can be renamed without affecting the tables on which the view is based.
- Isolate applications from changes in definitions of base tables

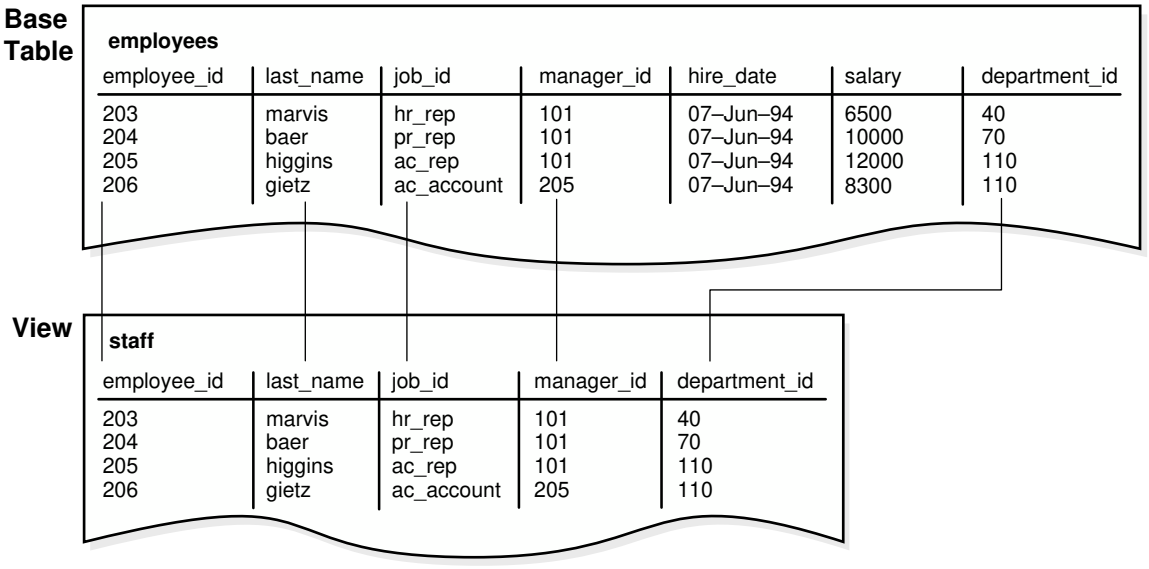
  For example, if the defining query of a view references three columns of a four column table, and a fifth column is added to the table, then the definition of the view is not affected, and all applications using the view are not affected.

For an example of the use of views, consider the `hr.employees` table, which has several columns and numerous rows. To allow users to see only five of these columns or only specific rows, you could create a view as follows:

```
CREATE VIEW staff AS
  SELECT employee_id, last_name, job_id, manager_id, department_id
  FROM   employees;
```

As with all subqueries, the query that defines a view cannot contain the `FOR UPDATE` clause. The following graphic illustrates the view named `staff`. Notice that the view shows only five of the columns in the base table.

**Figure 6-7 View**




- Characteristics of Views
  Unlike a table, a view is not allocated storage space, nor does a view contain data. Rather, a view is defined by a query that extracts or derives data from the base tables referenced by the view. Because a view is based on other objects, it requires no storage other than storage for the query that defines the view in the data dictionary.
- Updatable Join Views
  A **join view** has multiple tables or views in its `FROM` clause.
- Object Views
  Just as a view is a virtual table, an **object view** is a virtual object table. Each row in the view is an object, which is an instance of an **object type**. An object type is a user-defined data type.

> **See Also:**
>
> - "Overview of Materialized Views"
> - *Oracle Database Administrator's Guide* to learn how to manage views
> - *Oracle Database SQL Language Reference* for `CREATE VIEW` syntax and semantics

## Characteristics of Views

Unlike a table, a view is not allocated storage space, nor does a view contain data. Rather, a view is defined by a query that extracts or derives data from the base tables referenced by the

view. Because a view is based on other objects, it requires no storage other than storage for the query that defines the view in the data dictionary.

A view has dependencies on its referenced objects, which are automatically handled by the database. For example, if you drop and re-create a base table of a view, then the database determines whether the new base table is acceptable to the view definition.

- Data Manipulation in Views
  Because views are derived from tables, they have many similarities. Users can query views, and with some restrictions they can perform DML on views. Operations performed on a view affect data in some base table of the view and are subject to the integrity constraints and triggers of the base tables.
- How Data Is Accessed in Views
  Oracle Database stores a view definition in the data dictionary as the text of the query that defines the view.

## Data Manipulation in Views

Because views are derived from tables, they have many similarities. Users can query views, and with some restrictions they can perform DML on views. Operations performed on a view affect data in some base table of the view and are subject to the integrity constraints and triggers of the base tables.

The following example creates a view of the `hr.employees` table:

```
CREATE VIEW staff_dept_10 AS
SELECT employee_id, last_name, job_id,
       manager_id, department_id
FROM   employees
WHERE  department_id = 10
WITH CHECK OPTION CONSTRAINT staff_dept_10_cnst;
```

The defining query references only rows for department 10. The `CHECK OPTION` creates the view with a constraint so that `INSERT` and `UPDATE` statements issued against the view cannot result in rows that the view cannot select. Thus, rows for employees in department 10 can be inserted, but not rows for department 30.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about subquery restrictions in `CREATE VIEW` statements

## How Data Is Accessed in Views

Oracle Database stores a view definition in the data dictionary as the text of the query that defines the view.

When you reference a view in a SQL statement, Oracle Database performs the following tasks:

1. Merges a query (whenever possible) against a view with the queries that define the view and any underlying views

   Oracle Database optimizes the merged query as if you issued the query without referencing the views. Therefore, Oracle Database can use indexes on any referenced

base table columns, whether the columns are referenced in the view definition or in the user query against the view.

Sometimes Oracle Database cannot merge the view definition with the user query. In such cases, Oracle Database may not use all indexes on referenced columns.

2. Parses the merged statement in a shared SQL area

   Oracle Database parses a statement that references a view in a new shared SQL area *only* if no existing shared SQL area contains a similar statement. Thus, views provide the benefit of reduced memory use associated with shared SQL.

3. Executes the SQL statement

The following example illustrates data access when a view is queried. Assume that you create `employees_view` based on the `employees` and `departments` tables:

```
CREATE VIEW employees_view AS
  SELECT employee_id, last_name, salary, location_id
  FROM   employees JOIN departments USING (department_id)
  WHERE  department_id = 10;
```

A user executes the following query of `employees_view`:

```
SELECT last_name
FROM   employees_view
WHERE  employee_id = 200;
```

Oracle Database merges the view and the user query to construct the following query, which it then executes to retrieve the data:

```
SELECT last_name
FROM   employees, departments
WHERE  employees.department_id = departments.department_id
AND    departments.department_id = 10
AND    employees.employee_id = 200;
```

**See Also:**

- "Shared SQL Areas"
- "Overview of the Optimizer"
- *Oracle Database SQL Tuning Guide* to learn about query optimization

## Updatable Join Views

A **join view** has multiple tables or views in its `FROM` clause.

In the following example, the `staff_dept_10_30` view joins the `employees` and `departments` tables, including only employees in departments 10 or 30:

```
CREATE VIEW staff_dept_10_30 AS
SELECT employee_id, last_name, job_id, e.department_id
FROM   employees e, departments d
WHERE  e.department_id IN (10, 30)
AND    e.department_id = d.department_id;
```

An updatable join view, also called a *modifiable join view*, involves two or more base tables or views and permits DML operations. An updatable view contains multiple tables in the top-level `FROM` clause of the `SELECT` statement and is not restricted by the `WITH READ ONLY` clause.

To be inherently updatable, a view must meet several criteria. For example, a general rule is that an `INSERT`, `UPDATE`, or `DELETE` operation on a join view can modify only one base table at a time. The following query of the `USER_UPDATABLE_COLUMNS` data dictionary view shows that the `staff_dept_10_30` view is updatable:

```
SQL> SELECT TABLE_NAME, COLUMN_NAME, UPDATABLE
  2  FROM   USER_UPDATABLE_COLUMNS
  3  WHERE  TABLE_NAME = 'STAFF_DEPT_10_30';

TABLE_NAME                     COLUMN_NAME                    UPD
------------------------------ ------------------------------ ---
STAFF_DEPT_10_30               EMPLOYEE_ID                    YES
STAFF_DEPT_10_30               LAST_NAME                      YES
STAFF_DEPT_10_30               JOB_ID                         YES
STAFF_DEPT_10_30               DEPARTMENT_ID                  YES
```

Starting with Oracle Database Release 21c, it is not mandatory for all updatable columns in a join view to map to columns of a key-preserved table. The key-preserved table is a table in which each row of the underlying table appears at most once in the query output. In the `staff_dept_10_30` view, `department_id` is the primary key of the `departments` table, so each row from the `employees` table appears at most once in the result set, making the `employees` table key-preserved. The `departments` table is not key-preserved because each of its rows may appear many times in the result set. The columns in a non-key-preserved table can be updated if the `UPDATE` operation only updates columns from a single table and the update is deterministic, that is, it updates each row only once.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to update join views

## Object Views

Just as a view is a virtual table, an **object view** is a virtual object table. Each row in the view is an object, which is an instance of an **object type**. An object type is a user-defined data type.

You can retrieve, update, insert, and delete relational data as if it were stored as an object type. You can also define views with columns that are object data types, such as objects, `REF`s, and collections (nested tables and `VARRAY`s).

Like relational views, object views can present only the data that database administrators want users to see. For example, an object view could present data about IT programmers but omit sensitive data about salaries. The following example creates an `employee_type` object and then the view `it_prog_view` based on this object:

```
CREATE TYPE employee_type AS OBJECT
(
  employee_id  NUMBER (6),
  last_name    VARCHAR2 (25),
  job_id       VARCHAR2 (10)
);
/

CREATE VIEW it_prog_view OF employee_type
  WITH OBJECT IDENTIFIER (employee_id) AS
SELECT  e.employee_id, e.last_name, e.job_id
FROM    employees e
WHERE   job_id = 'IT_PROG';
```

Object views are useful in prototyping or transitioning to object-oriented applications because the data in the view can be taken from relational tables and accessed as if the table were defined as an object table. You can run object-oriented applications without converting existing tables to a different physical structure.

**See Also:**

- *Oracle Database Object-Relational Developer's Guide* to learn about object types and object views
- *Oracle Database SQL Language Reference* to learn about the `CREATE TYPE` statement

# Overview of Materialized Views

A **materialized view** is a query result that has been stored or "materialized" in advance as a schema object. The `FROM` clause of the query can name tables, views, or materialized views.

A materialized view often serves as a master table in replication and a fact table in data warehousing. Materialized views summarize, compute, replicate, and distribute data. They are suitable in various computing environments, such as the following:

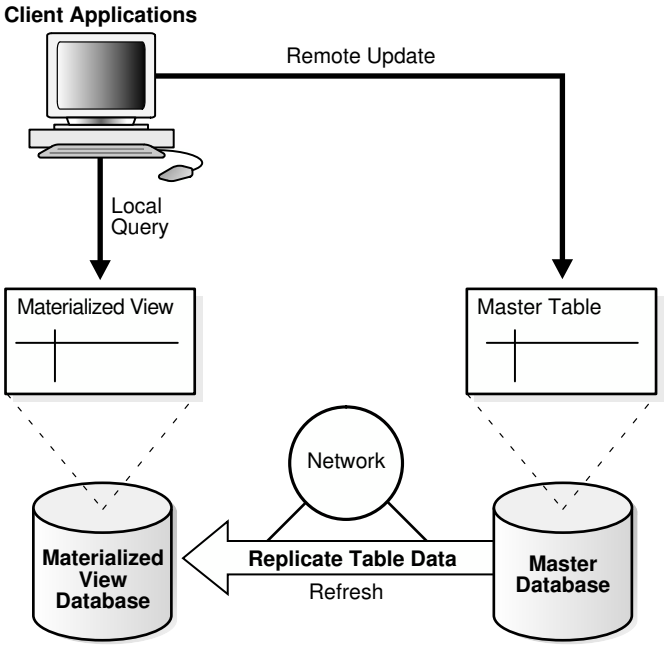
- In data warehouses, materialized views can compute and store data generated from aggregate functions such as sums and averages.

  A summary is an aggregate view that reduces query time by precalculating joins and aggregation operations and storing the results in a table. Materialized views are equivalent to summaries. You can also use materialized views to compute joins with or without aggregations.
- In materialized view replication, which is achieved using XStream and Oracle GoldenGate, the view contains a complete or partial copy of a table from a single point in time. Materialized views replicate data at distributed sites and synchronize updates performed at several sites. This form of replication is suitable for environments such as field sales when databases are not always connected to the network.

- In mobile computing environments, materialized views can download a data subset from central servers to mobile clients, with periodic refreshes from the central servers and propagation of updates by clients to the central servers.

In a replication environment, a materialized view shares data with a table in a different database, called a master database. The table associated with the materialized view at the master site is the master table. Figure 6-8 illustrates a materialized view in one database based on a master table in another database. Updates to the master table replicate to the materialized view database.

**Figure 6-8 Materialized View**




- Characteristics of Materialized Views
  Materialized views share some characteristics of indexes and nonmaterialized views.
- Refresh Methods for Materialized Views
  The database maintains data in materialized views by refreshing them after changes to the base tables. The refresh method can be incremental or a complete refresh.
- Automatic Materialized Views
  Starting with Oracle Database Release 21c, materialized views can be created and maintained automatically.
- Query Rewrite
  **Query rewrite** transforms a user request written in terms of master tables into a semantically equivalent request that includes materialized views.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* to learn more about summaries
> - *Oracle Database XStream Guide* for an introduction to XStream
> - `http://www.oracle.com/technetwork/middleware/goldengate/documentation/index.html` to learn more about Oracle GoldenGate
> - *Oracle Database SQL Language Reference* to learn about the `CREATE MATERIALIZED VIEW` statement

## Characteristics of Materialized Views

Materialized views share some characteristics of indexes and nonmaterialized views.

Materialized views are similar to indexes in the following ways:

- They contain actual data and consume storage space.
- They can be refreshed when the data in their master tables changes.
- They can improve performance of SQL execution when used for query rewrite operations.
- Their existence is transparent to SQL applications and users.

A materialized view is similar to a nonmaterialized view because it represents data in other tables and views. Unlike indexes, users can query materialized views directly using `SELECT` statements. Depending on the types of refresh that are required, the views can also be updated with DML statements.

The following example creates and populates a materialized aggregate view based on three master tables in the `sh` sample schema:

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

The following example drops table `sales`, which is a master table for `sales_mv`, and then queries `sales_mv`. The query selects data because the rows are stored (materialized) separately from the data in the master tables.

```
SQL> DROP TABLE sales;

Table dropped.

SQL> SELECT * FROM sales_mv WHERE ROWNUM < 4;

CALENDAR_YEAR    PROD_ID  SUM_SALES
------------- ---------- ----------
         1998         13  936197.53
```

```
1998         26  567533.83
1998         27  107968.24
```

A materialized view can be partitioned. You can define a materialized view on a partitioned table and one or more indexes on the materialized view.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* to learn how to use materialized views in a data warehouse

# Refresh Methods for Materialized Views

The database maintains data in materialized views by refreshing them after changes to the base tables. The refresh method can be incremental or a complete refresh.

- Complete Refresh
  A **complete refresh** executes the query that defines the materialized view. A complete refresh occurs when you initially create the materialized view, unless the materialized view references a prebuilt table, or you define the table as `BUILD DEFERRED`.
- Incremental Refresh
  An **incremental refresh**, also called a *fast refresh*, processes only the changes to the existing data. This method eliminates the need to rebuild materialized views from the beginning. Processing only the changes can result in a very fast refresh time.
- In-Place and Out-of-Place Refresh
  For the complete and incremental methods, the database can refresh the materialized view in place, which refreshes statements directly on the view, or out of place.

## Complete Refresh

A **complete refresh** executes the query that defines the materialized view. A complete refresh occurs when you initially create the materialized view, unless the materialized view references a prebuilt table, or you define the table as `BUILD DEFERRED`.

A complete refresh can be slow, especially if the database must read and process huge amounts of data. You can perform a complete refresh at any time after creation of the materialized view.

## Incremental Refresh

An **incremental refresh**, also called a *fast refresh*, processes only the changes to the existing data. This method eliminates the need to rebuild materialized views from the beginning. Processing only the changes can result in a very fast refresh time.

You can refresh materialized views either on demand or at regular time intervals. Alternatively, you can configure materialized views in the same database as their base tables to refresh whenever a transaction commits changes to the base tables.

Fast refresh comes in either of the following forms:

- Log-Based refresh

In this type of refresh, a materialized view log or a direct loader log keeps a record of changes to the base tables. A materialized view log is a schema object that records changes to a base table so that a materialized view defined on the base table can be refreshed incrementally. Each materialized view log is associated with a single base table.

- Partition change tracking (PCT) refresh

  PCT refresh is valid only when the base tables are partitioned. PCT refresh removes all data in the affected materialized view partitions or affected portions of data, and then recomputes them. The database uses the modified base table partitions to identify the affected partitions or portions of data in the view. When partition maintenance operations have occurred on the base tables, PCT refresh is the only usable incremental refresh method.

## In-Place and Out-of-Place Refresh

For the complete and incremental methods, the database can refresh the materialized view in place, which refreshes statements directly on the view, or out of place.

An out-of-place refresh creates one or more outside tables, executes the refresh statements on them, and then switches the materialized view or affected partitions with the outside tables. This technique achieves high availability during refresh, especially when refresh statements take a long time to finish.

Synchronous refresh is a type of out-of-place refresh. A synchronous refresh does not modify the contents of the base tables, but instead uses the APIs in the synchronous refresh package, which ensures consistency by applying these changes to the base tables and materialized views at the same time. This approach enables a set of tables and the materialized views defined on them to be always synchronized. In a data warehouse, synchronous refresh method is well-suited for the following reasons:

- The loading of incremental data is tightly controlled and occurs at periodic intervals.
- Tables and their materialized views are often partitioned in the same way, or their partitions are related by a functional dependency.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* to learn how to refresh materialized views

# Automatic Materialized Views

Starting with Oracle Database Release 21c, materialized views can be created and maintained automatically.

Oracle Database can automatically create and manage materialized views in order to optimize query performance. With very little or no interaction with the DBA, background tasks monitor and analyze workload characteristics and identifies where materialized views will improve SQL performance. The performance benefit of candidate materialized views is measured in the background (using workload queries) before they are made visible to the workload.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for additional information
> - *Oracle Database PL/SQL Packages and Types Reference* to learn how to use the `DBMS_AUTO_MV` package to implement automatic materialized views

## Query Rewrite

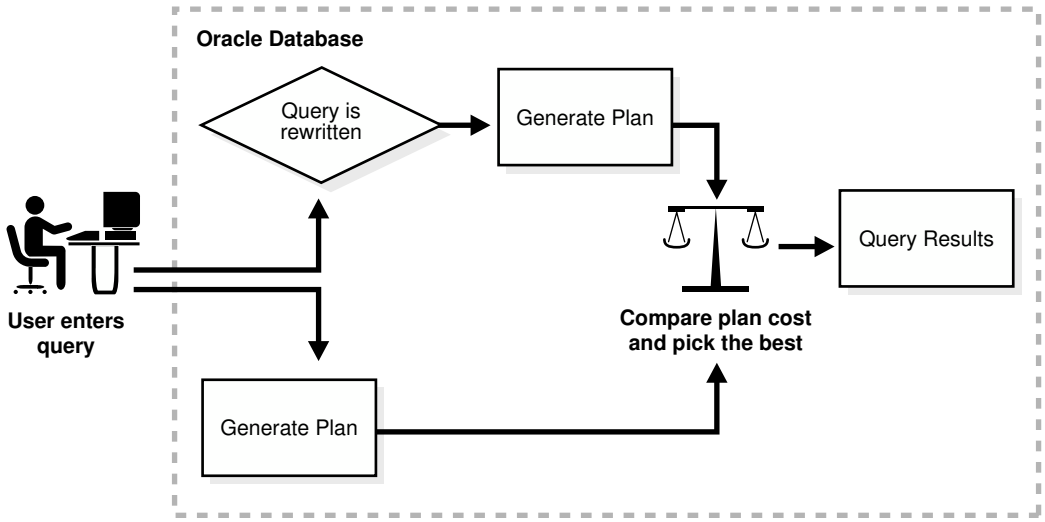
**Query rewrite** transforms a user request written in terms of master tables into a semantically equivalent request that includes materialized views.

When base tables contain large amounts of data, computing an aggregate or join is expensive and time-consuming. Because materialized views contain precomputed aggregates and joins, query rewrite can quickly answer queries using materialized views.

The query transformer transparently rewrites the request to use the materialized view, requiring no user intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped without invalidating the SQL in the application code.

In general, rewriting queries to use materialized views rather than detail tables improves response time. The following figure shows the database generating an execution plan for the original and rewritten query and choosing the lowest-cost plan.

**Figure 6-9 Query Rewrite**




> **See Also:**
>
> - "Overview of the Optimizer" to learn more about query transformation
> - *Oracle Database Data Warehousing Guide* to learn how to use query rewrite

# Overview of Sequences

A **sequence** is a schema object from which multiple users can generate unique integers. A sequence generator provides a highly scalable and well-performing method to generate surrogate keys for a number data type.

- Sequence Characteristics
  A sequence definition indicates general information about the sequence, including its name and whether the sequence ascends or descends.
- Concurrent Access to Sequences
  The same sequence generator can generate numbers for multiple tables.

## Sequence Characteristics

A sequence definition indicates general information about the sequence, including its name and whether the sequence ascends or descends.

A sequence definition also indicates:

- The interval between numbers
- Whether the database should cache sets of generated sequence numbers in memory
- Whether the sequence should cycle when a limit is reached

The following example creates the sequence `customers_seq` in the sample schema `oe`. An application could use this sequence to provide customer ID numbers when rows are added to the `customers` table.

```
CREATE SEQUENCE customers_seq
START WITH      1000
INCREMENT BY    1
NOCACHE
NOCYCLE;
```

The first reference to `customers_seq.nextval` returns `1000`. The second returns `1001`. Each subsequent reference returns a value 1 greater than the previous reference.

> **See Also:**
>
> - *Oracle Database 2 Day Developer's Guide* for a tutorial that shows you how to create a sequence
> - *Oracle Database Administrator's Guide* to learn how to reference a sequence in a SQL statement
> - *Oracle Database SQL Language Reference* for `CREATE SEQUENCE` syntax and semantics

## Concurrent Access to Sequences

The same sequence generator can generate numbers for multiple tables.

The generator can create primary keys automatically and coordinate keys across multiple rows or tables. For example, a sequence can generate primary keys for an `orders` table and a `customers` table.

The sequence generator is useful in multiuser environments for generating unique numbers without the overhead of disk I/O or transaction locking. For example, two users simultaneously insert new rows into the `orders` table. By using a sequence to generate unique numbers for the `order_id` column, neither user has to wait for the other to enter the next available order number. The sequence automatically generates the correct values for each user.

Each user that references a sequence has access to his or her current sequence number, which is the last sequence generated in the session. A user can issue a statement to generate a new sequence number or use the current number last generated by the session. After a statement in a session generates a sequence number, it is available only to this session. Individual sequence numbers can be skipped if they were generated and used in a transaction that was ultimately rolled back.

> **WARNING:**
>
> If your application requires a gap-free set of numbers, then you cannot use Oracle sequences. You must serialize activities in the database using your own developed code.

> **See Also:**
>
> "Data Concurrency and Consistency " to learn how sessions access data at the same time

# Overview of Dimensions

A typical data warehouse has two important components: dimensions and facts.

A **dimension** is any category used in specifying business questions, for example, time, geography, product, department, and distribution channel. A **fact** is an event or entity associated with a particular set of dimension values, for example, units sold or profits.

Examples of multidimensional requests include the following:

- Show total sales across all products at increasing aggregation levels for a geography dimension, from state to country to region, for 2013 and 2014.
- Create a cross-tabular analysis of our operations showing expenses by territory in South America for 2013 and 2014. Include all possible subtotals.
- List the top 10 sales representatives in Asia according to 2014 sales revenue for automotive products, and rank their commissions.

Many multidimensional questions require aggregated data and comparisons of data sets, often across time, geography or budgets.

Creating a dimension permits the broader use of the query rewrite feature. By transparently rewriting queries to use materialized views, the database can improve query performance.

- Hierarchical Structure of a Dimension
  A **dimension table** is a logical structure that defines hierarchical (parent/child) relationships between pairs of columns or column sets.
- Creation of Dimensions
  You create dimensions with the `CREATE DIMENSION` SQL statement.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* to learn more about dimensions

## Hierarchical Structure of a Dimension

A **dimension table** is a logical structure that defines hierarchical (parent/child) relationships between pairs of columns or column sets.

For example, a dimension can indicate that within a row the `city` column implies the value of the `state` column, and the `state` column implies the value of the `country` column.

Within a customer dimension, customers could roll up to city, state, country, subregion, and region. Data analysis typically starts at higher levels in the dimensional hierarchy and gradually drills down if the situation warrants such analysis.

Each value at the child level is associated with one and only one value at the parent level. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next level in the hierarchy.

A dimension has no data storage assigned to it. Dimensional information is stored in dimension tables, whereas fact information is stored in a fact table.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* to learn about dimensions
> - *Oracle OLAP User's Guide* to learn how to create dimensions

## Creation of Dimensions

You create dimensions with the `CREATE DIMENSION` SQL statement.

This statement specifies:

- Multiple `LEVEL` clauses, each of which identifies a column or column set in the dimension
- One or more `HIERARCHY` clauses that specify the parent/child relationships between adjacent levels
- Optional `ATTRIBUTE` clauses, each of which identifies an additional column or column set associated with an individual level

The following statement was used to create the `customers_dim` dimension in the sample schema `sh`:

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name);
```

The columns in a dimension can come either from the same table (denormalized) or from multiple tables (fully or partially normalized). For example, a normalized time dimension can include a date table, a month table, and a year table, with join conditions that connect each date row to a month row, and each month row to a year row. In a fully denormalized time dimension, the date, month, and year columns are in the same table. Whether normalized or denormalized, the hierarchical relationships among the columns must be specified in the `CREATE DIMENSION` statement.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `CREATE DIMENSION` syntax and semantics

# Overview of Synonyms

A **synonym** is an alias for a schema object. For example, you can create a synonym for a table or view, sequence, PL/SQL program unit, user-defined object type, or another synonym. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms can simplify SQL statements for database users. Synonyms are also useful for hiding the identity and location of an underlying schema object. If the underlying object must be renamed or moved, then only the synonym must be redefined. Applications based on the synonym continue to work without modification.

You can create both private and public synonyms. A private synonym is in the schema of a specific user who has control over its availability to others. A public synonym is owned by the user group named `PUBLIC` and is accessible by every database user.

### Example 6-5 Public Synonym

Suppose that a database administrator creates a public synonym named `people` for the `hr.employees` table. The user then connects to the `oe` schema and counts the number of rows in the table referenced by the synonym.

```
SQL> CREATE PUBLIC SYNONYM people FOR hr.employees;

Synonym created.

SQL> CONNECT oe
Enter password: password
Connected.

SQL> SELECT COUNT(*) FROM people;

  COUNT(*)
----------
       107
```

Use public synonyms sparingly because they make database consolidation more difficult. As shown in the following example, if another administrator attempts to create the public synonym `people`, then the creation fails because only one public synonym `people` can exist in the database. Overuse of public synonyms causes namespace conflicts between applications.

```
SQL> CREATE PUBLIC SYNONYM people FOR oe.customers;
CREATE PUBLIC SYNONYM people FOR oe.customers
                     *
ERROR at line 1:
ORA-00955: name is already used by an existing object

SQL> SELECT OWNER, SYNONYM_NAME, TABLE_OWNER, TABLE_NAME
  2  FROM DBA_SYNONYMS
  3  WHERE SYNONYM_NAME = 'PEOPLE';

OWNER      SYNONYM_NAME TABLE_OWNER TABLE_NAME
---------- ------------ ----------- ----------
PUBLIC     PEOPLE       HR          EMPLOYEES
```

Synonyms themselves are not securable. When you grant object privileges on a synonym, you are really granting privileges on the underlying object. The synonym is acting only as an alias for the object in the `GRANT` statement.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to manage synonyms
> - *Oracle Database SQL Language Reference* for `CREATE SYNONYM` syntax and semantics

7

# Data Integrity

This chapter explains how integrity constraints enforce the business rules associated with a database and prevent the entry of invalid information into tables.

This chapter contains the following sections:


- Introduction to Data Integrity
- Types of Integrity Constraints
- States of Integrity Constraints


- Introduction to Data Integrity
  It is important that data maintain **data integrity**, which is adherence to business rules determined by the database administrator or application developer.
- Types of Integrity Constraints
  Oracle Database enables you to apply constraints both at the table and column level.
- States of Integrity Constraints
  As part of constraint definition, you can specify how and when Oracle Database should enforce the constraint, thereby determining the constraint state.

> **See Also:**
>
> "Overview of Tables" for background on columns and the need for integrity constraints

## Introduction to Data Integrity

It is important that data maintain **data integrity**, which is adherence to business rules determined by the database administrator or application developer.

Business rules specify conditions and relationships that must always be true or must always be false. For example, each company defines its own policies about salaries, employee numbers, inventory tracking, and so on.

- Techniques for Guaranteeing Data Integrity
  When designing a database application, developers have several options for guaranteeing the integrity of data stored in the database.
- Advantages of Integrity Constraints
  An integrity constraint is a schema object that is created and dropped using SQL. To enforce data integrity, use integrity constraints whenever possible.

### Techniques for Guaranteeing Data Integrity

When designing a database application, developers have several options for guaranteeing the integrity of data stored in the database.

These options include:

- Enforcing business rules with triggered stored database procedures
- Using stored procedures to completely control access to data
- Enforcing business rules in the code of a database application
- Using Oracle Database integrity constraints, which are rules defined at the column or object level that restrict values in the database

**See Also:**

- "Overview of Triggers" explains the purpose and types of triggers
- "Introduction to Server-Side Programming" explains the purpose and characteristics of stored procedures

## Advantages of Integrity Constraints

An integrity constraint is a schema object that is created and dropped using SQL. To enforce data integrity, use integrity constraints whenever possible.

Advantages of integrity constraints over alternatives for enforcing data integrity include:

- Declarative ease

  Because you define integrity constraints using SQL statements, no additional programming is required when you define or alter a table. The SQL statements are easy to write and eliminate programming errors.

- Centralized rules

  Integrity constraints are defined for tables and are stored in the data dictionary. Thus, data entered by all applications must adhere to the same integrity constraints. If the rules change at the table level, then applications need not change. Also, applications can use metadata in the data dictionary to immediately inform users of violations, even before the database checks the SQL statement.

- Flexibility when loading data

  You can disable integrity constraints temporarily to avoid performance overhead when loading large amounts of data. When the data load is complete, you can re-enable the integrity constraints.

**See Also:**

- "Overview of the Data Dictionary"
- *Oracle Database 2 Day Developer's Guide* and *Oracle Database Development Guide* to learn how to maintain data integrity
- *Oracle Database Administrator's Guide* to learn how to manage integrity constraints

# Types of Integrity Constraints

Oracle Database enables you to apply constraints both at the table and column level.

A constraint specified as part of the definition of a column or attribute is an inline specification. A constraint specified as part of the table definition is an out-of-line specification.

A key is the column or set of columns included in the definition of certain types of integrity constraints. Keys describe the relationships between the tables and columns of a relational database. Individual values in a key are called key values.

The following table describes the types of constraints. Each can be specified either inline or out-of-line, except for `NOT NULL`, which must be inline.

**Table 7-1 Types of Integrity Constraints**

| Constraint Type | Description | See Also |
|---|---|---|
| `NOT NULL` | Allows or disallows inserts or updates of rows containing a null in a specified column. | "NOT NULL Integrity Constraints" |
| Unique key | Prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null. | "Unique Constraints" |
| Primary key | Combines a `NOT NULL` constraint and a unique constraint. It prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null. | "Primary Key Constraints" |
| Foreign key | Designates a column as the foreign key and establishes a relationship between the foreign key and a primary or unique key, called the referenced key. | "Foreign Key Constraints" |
| Check | Requires a database value to obey a specified condition. | "Check Constraints" |
| `REF` | Dictates types of data manipulation allowed on values in a `REF` column and how these actions affect dependent values. In an object-relational database, a built-in data type called a `REF` encapsulates a reference to a row object of a specified object type. Referential integrity constraints on `REF` columns ensure that there is a row object for the `REF`. | *Oracle Database Object-Relational Developer's Guide* to learn about `REF` constraints |

- NOT NULL Integrity Constraints
  A `NOT NULL` constraint requires that a column of a table contain no null values. A **null** is the absence of a value. By default, all columns in a table allow nulls.
- Unique Constraints
  A **unique key constraint** requires that every value in a column or set of columns be unique. No rows of a table may have duplicate values in a single column (the **unique key**) or set of columns (the **composite unique key**) with a unique key constraint.

- Primary Key Constraints
  In a **primary key constraint**, the values in the group of one or more columns subject to the constraint uniquely identify the row. Each table can have one **primary key**, which in effect names the row and ensures that no duplicate rows exist.
- Foreign Key Constraints
  Whenever two tables contain one or more common columns, Oracle Database can enforce the relationship between the two tables through a **foreign key constraint**, also called a *referential integrity constraint*.
- Check Constraints
  A **check constraint** on a column or set of columns requires that a specified **condition** be true or unknown for every row.

> **See Also:**
>
> - "Overview of Tables"
> - *Oracle Database SQL Language Reference* to learn more about the types of constraints

## NOT NULL Integrity Constraints

A `NOT NULL` constraint requires that a column of a table contain no null values. A **null** is the absence of a value. By default, all columns in a table allow nulls.

`NOT NULL` constraints are intended for columns that must not lack values. For example, the `hr.employees` table requires a value in the `email` column. An attempt to insert an employee row without an email address generates an error:

```
SQL> INSERT INTO hr.employees (employee_id, last_name) values (999, 'Smith');
.
.
.
ERROR at line 1:
ORA-01400: cannot insert NULL into ("HR"."EMPLOYEES"."EMAIL")
```

You can only add a column with a `NOT NULL` constraint if the table does not contain any rows or if you specify a default value.

> **See Also:**
>
> - *Oracle Database 2 Day Developer's Guide* for examples of adding `NOT NULL` constraints to a table
> - *Oracle Database SQL Language Reference* for restrictions on using `NOT NULL` constraints
> - *Oracle Database Development Guide* to learn when to use the `NOT NULL` constraint

# Unique Constraints

A **unique key constraint** requires that every value in a column or set of columns be unique. No rows of a table may have duplicate values in a single column (the **unique key**) or set of columns (the **composite unique key**) with a unique key constraint.

> **Note:**
>
> The term *key* refers only to the columns defined in the integrity constraint. Because the database enforces a unique constraint by implicitly creating or reusing an index on the key columns, the term *unique key* is sometimes incorrectly used as a synonym for *unique key constraint* or *unique index*.

Unique key constraints are appropriate for any column where duplicate values are not allowed. Unique constraints differ from primary key constraints, whose purpose is to identify each table row uniquely, and typically contain values that have no significance other than being unique. Examples of unique keys include:

- A customer phone number, where the primary key is the customer number
- A department name, where the primary key is the department number

As shown in Example 4-1, a unique key constraint exists on the `email` column of the `hr.employees` table. The relevant part of the statement is as follows:

```
CREATE TABLE employees    ( ...
    , email          VARCHAR2(25)
        CONSTRAINT   emp_email_nn  NOT NULL ...
    , CONSTRAINT     emp_email_uk  UNIQUE (email) ... );
```

The `emp_email_uk` constraint ensures that no two employees have the same email address, as shown in the following example:

```
SQL> SELECT employee_id, last_name, email FROM employees WHERE email = 'PFAY';

EMPLOYEE_ID LAST_NAME                 EMAIL
----------- ------------------------- -------------------------
        202 Fay                       PFAY

SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id)
  1  VALUES (999,'Fay','PFAY',SYSDATE,'ST_CLERK');
.
.
.
ERROR at line 1:
ORA-00001: unique constraint (HR.EMP_EMAIL_UK) violated
```

Unless a `NOT NULL` constraint is also defined, a null always satisfies a unique key constraint. Thus, columns with both unique key constraints and `NOT NULL` constraints are typical. This combination forces the user to enter values in the unique key and eliminates the possibility that new row data conflicts with existing row data.

> **Note:**
>
> Because of the search mechanism for unique key constraints on multiple columns, you cannot have identical values in the non-null columns of a partially null composite unique key constraint.

**Example 7-1 Unique Constraint**

```
SQL> SELECT employee_id, last_name, email FROM employees WHERE email = 'PFAY';

EMPLOYEE_ID LAST_NAME                 EMAIL
----------- ------------------------- -------------------------
        202 Fay                       PFAY

SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id)
  1  VALUES (999,'Fay','PFAY',SYSDATE,'ST_CLERK');
.
.
.
ERROR at line 1:
ORA-00001: unique constraint (HR.EMP_EMAIL_UK) violated
```

> **See Also:**
>
> - "Unique and Nonunique Indexes"
> - *Oracle Database 2 Day Developer's Guide* for examples of adding `UNIQUE` constraints to a table

# Primary Key Constraints

In a **primary key constraint**, the values in the group of one or more columns subject to the constraint uniquely identify the row. Each table can have one **primary key**, which in effect names the row and ensures that no duplicate rows exist.

A primary key can be natural or a surrogate. A natural key is a meaningful identifier made of existing attributes in a table. For example, a natural key could be a postal code in a lookup table. In contrast, a surrogate key is a system-generated incrementing identifier that ensures uniqueness within a table. Typically, a sequence generates surrogate keys.

The Oracle Database implementation of the primary key constraint guarantees that the following statements are true:

- No two rows have duplicate values in the specified column or set of columns.
- The primary key columns do not allow nulls.

A typical situation calling for a primary key is the numeric identifier for an employee. Each employee must have a unique ID. An employee must be described by one and only one row in the `employees` table.

The example in Unique Constraints indicates that an existing employee has the employee ID of 202, where the employee ID is the primary key. The following example shows an attempt to add an employee with the same employee ID and an employee with no ID:

```
SQL> INSERT INTO employees (employee_id, last_name, email, hire_date,
job_id)
  1  VALUES (202,'Chan','JCHAN',SYSDATE,'ST_CLERK');
.
.
.
ERROR at line 1:
ORA-00001: unique constraint (HR.EMP_EMP_ID_PK) violated

SQL> INSERT INTO employees (last_name) VALUES ('Chan');
.
.
.
ERROR at line 1:
ORA-01400: cannot insert NULL into ("HR"."EMPLOYEES"."EMPLOYEE_ID")
```

The database enforces primary key constraints with an index. Usually, a primary key constraint created for a column implicitly creates a unique index and a `NOT NULL` constraint. Note the following exceptions to this rule:

- In some cases, as when you create a primary key with a deferrable constraint, the generated index is not unique.

  > **Note:**
  >
  > You can explicitly create a unique index with the `CREATE UNIQUE INDEX` statement.

- If a usable index exists when a primary key constraint is created, then the constraint reuses this index and does not implicitly create one.

By default the name of the implicitly created index is the name of the primary key constraint. You can also specify a user-defined name for an index. You can specify storage options for the index by including the `ENABLE` clause in the `CREATE TABLE` or `ALTER TABLE` statement used to create the constraint.

> **See Also:**
>
> *Oracle Database 2 Day Developer's Guide* and *Oracle Database Development Guide* to learn how to add primary key constraints to a table

## Foreign Key Constraints

Whenever two tables contain one or more common columns, Oracle Database can enforce the relationship between the two tables through a **foreign key constraint**, also called a *referential integrity constraint*.

A foreign key constraint requires that for each value in the column on which the constraint is defined, the value in the other specified other table and column must match. An example of a referential integrity rule is an employee can work for only an existing department.

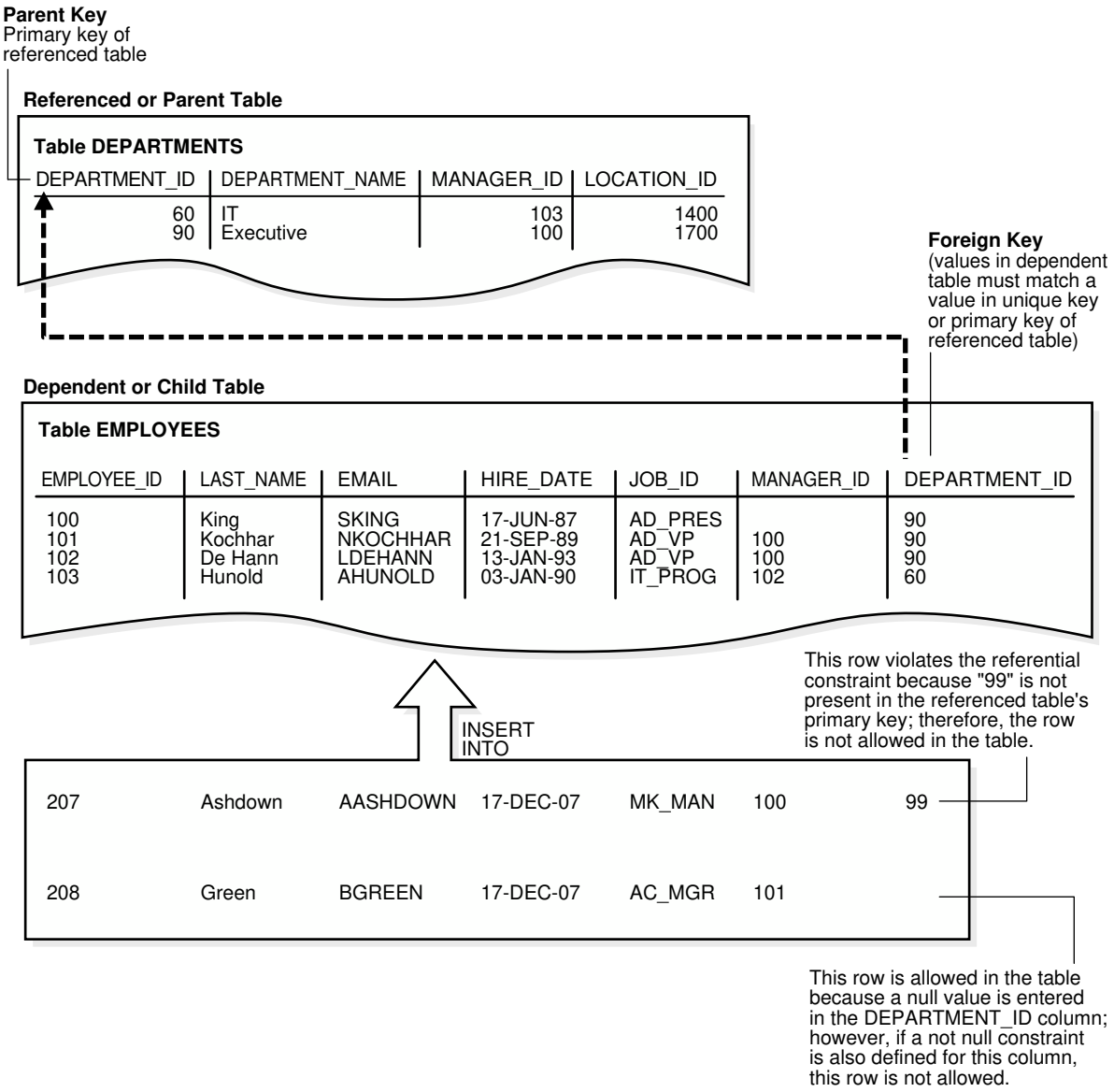
The following table lists terms associated with referential integrity constraints.

**Table 7-2 Referential Integrity Constraint Terms**

| Term | Definition |
|---|---|
| Foreign key | The column or set of columns included in the definition of the constraint that reference a referenced key. For example, the `department_id` column in `employees` is a foreign key that references the `department_id` column in `departments`.<br>Foreign keys may be defined as multiple columns. However, a composite foreign key must reference a composite primary or unique key with the same number of columns and the same data types.<br>The value of foreign keys can match either the referenced primary or unique key value, or be null. If any column of a composite foreign key is null, then the non-null portions of the key do not have to match any corresponding portion of a parent key. |
| Referenced key | The unique key or primary key of the table referenced by a foreign key. For example, the `department_id` column in `departments` is the referenced key for the `department_id` column in `employees`. |
| Dependent or child table | The table that includes the foreign key. This table depends on the values present in the referenced unique or primary key. For example, the `employees` table is a child of `departments`. |
| Referenced or parent table | The table that is referenced by the foreign key of the child table. It is this table's referenced key that determines whether specific inserts or updates are allowed in the child table. For example, the `departments` table is a parent of `employees`. |

Figure 7-1 shows a foreign key on the `employees.department_id` column. It guarantees that every value in this column must match a value in the `departments.department_id` column. Thus, no erroneous department numbers can exist in the `employees.department_id` column.

**Figure 7-1 Referential Integrity Constraints**




- Self-Referential Integrity Constraints
  A **self-referential integrity constraint** is a foreign key that references a parent key in the same table.
- Nulls and Foreign Keys
  The relational model permits the value of foreign keys to match either the referenced primary or unique key value, or be null. For example, a row in `hr.employees` might not specify a department ID.
- Parent Key Modifications and Foreign Keys
  The relationship between foreign key and parent key has implications for deletion of parent keys. For example, if a user attempts to delete the record for this department, then what happens to the records for employees in this department?

- Indexes and Foreign Keys
  As a rule, foreign keys should be indexed. The only exception is when the matching unique or primary key is never updated or deleted.

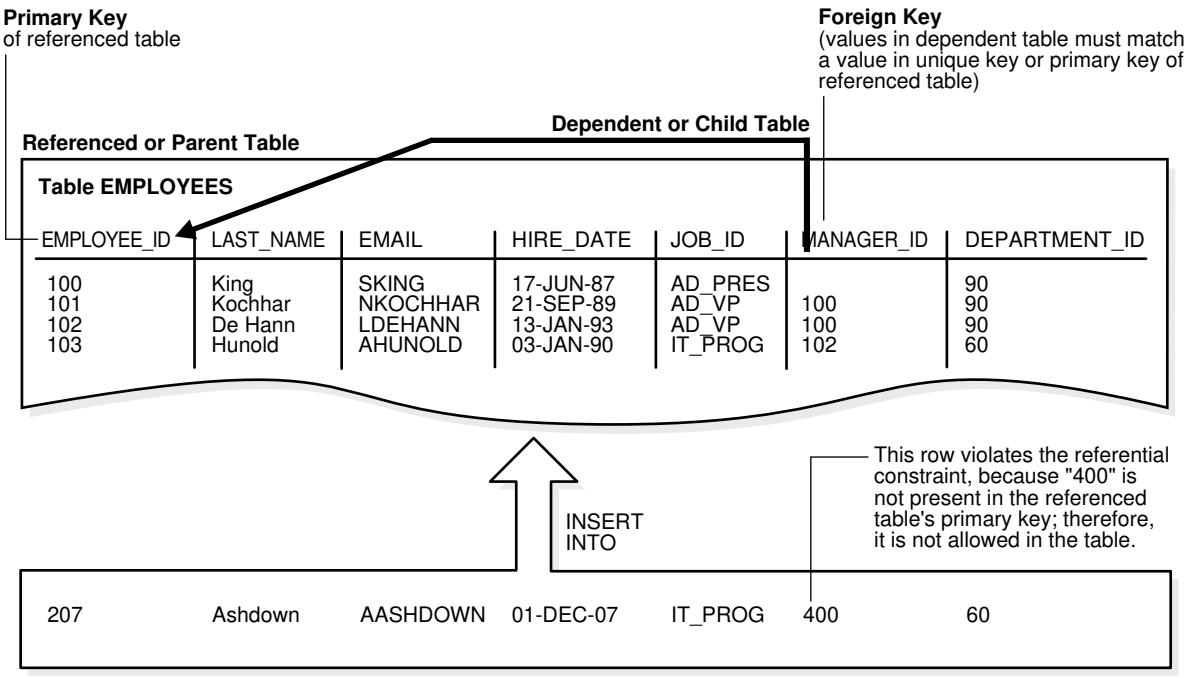
> **See Also:**
>
> *Oracle Database 2 Day Developer's Guide* and *Oracle Database Development Guide* to learn how to add foreign key constraints to a table

## Self-Referential Integrity Constraints

A **self-referential integrity constraint** is a foreign key that references a parent key in the same table.

In the following figure, a self-referential constraint ensures that every value in the `employees.manager_id` column corresponds to an existing value in the `employees.employee_id` column. For example, the manager for employee 102 must exist in the `employees` table. This constraint eliminates the possibility of erroneous employee numbers in the `manager_id` column.

**Figure 7-2 Single Table Referential Constraints**




## Nulls and Foreign Keys

The relational model permits the value of foreign keys to match either the referenced primary or unique key value, or be null. For example, a row in `hr.employees` might not specify a department ID.

If any column of a composite foreign key is null, then the non-null portions of the key do not have to match any corresponding portion of a parent key. For example, a `reservations` table might contain a composite foreign key on the `table_id` and `date` columns, but `table_id` is null.

## Parent Key Modifications and Foreign Keys

The relationship between foreign key and parent key has implications for deletion of parent keys. For example, if a user attempts to delete the record for this department, then what happens to the records for employees in this department?

When a parent key is modified, referential integrity constraints can specify the following actions to be performed on dependent rows in a child table:

- No action on deletion or update

  In the normal case, users cannot modify referenced key values if the results would violate referential integrity. For example, if `employees.department_id` is a foreign key to `departments`, and if employees belong to a particular department, then an attempt to delete the row for this department violates the constraint.

- Cascading deletions

  A deletion cascades (`DELETE CASCADE`) when rows containing referenced key values are deleted, causing all rows in child tables with dependent foreign key values to also be deleted. For example, the deletion of a row in `departments` causes rows for all employees in this department to be deleted.

- Deletions that set null

  A deletion sets null (`DELETE SET NULL`) when rows containing referenced key values are deleted, causing all rows in child tables with dependent foreign key values to set those values to null. For example, the deletion of a department row sets the `department_id` column value to null for employees in this department.

Table 7-3 outlines the DML statements allowed by the different referential actions on the key values in the parent table, and the foreign key values in the child table.

**Table 7-3 DML Statements Allowed by Update and Delete No Action**

| DML Statement | Issued Against Parent Table | Issued Against Child Table |
|---|---|---|
| `INSERT` | Always OK if the parent key value is unique | OK only if the foreign key value exists in the parent key or is partially or all null |
| `UPDATE NO ACTION` | Allowed if the statement does not leave any rows in the child table without a referenced parent key value | Allowed if the new foreign key value still references a referenced key value |
| `DELETE NO ACTION` | Allowed if no rows in the child table reference the parent key value | Always OK |
| `DELETE CASCADE` | Always OK | Always OK |
| `DELETE SET NULL` | Always OK | Always OK |

> **Note:**
>
> Other referential actions not supported by `FOREIGN KEY` integrity constraints of Oracle Database can be enforced using database triggers. See "Overview of Triggers".

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about the `ON DELETE` clause

## Indexes and Foreign Keys

As a rule, foreign keys should be indexed. The only exception is when the matching unique or primary key is never updated or deleted.

Indexing the foreign keys in child tables provides the following benefits:

- Prevents a full table lock on the child table. Instead, the database acquires a row lock on the index.
- Removes the need for a full table scan of the child table. As an illustration, assume that a user removes the record for department 10 from the `departments` table. If `employees.department_id` is not indexed, then the database must scan `employees` to see if any employees exist in department 10.

> **See Also:**
>
> - "Locks and Foreign Keys" explains the locking behavior for indexed and unindexed foreign key columns
> - "Introduction to Indexes" explains the purpose and characteristics of indexes

# Check Constraints

A **check constraint** on a column or set of columns requires that a specified **condition** be true or unknown for every row.

If DML results in the condition of the constraint evaluating to false, then the SQL statement is rolled back. The chief benefit of check constraints is the ability to enforce very specific integrity rules. For example, you could use check constraints to enforce the following rules in the `hr.employees` table:

- The `salary` column must not have a value greater than 10000.
- The `commission` column must have a value that is not greater than the salary.

The following example creates a maximum salary constraint on `employees` and demonstrates what happens when a statement attempts to insert a row containing a salary that exceeds the maximum:

```
SQL> ALTER TABLE employees ADD CONSTRAINT max_emp_sal CHECK (salary < 10001);
SQL> INSERT INTO employees
(employee_id,last_name,email,hire_date,job_id,salary)
  1  VALUES (999,'Green','BGREEN',SYSDATE,'ST_CLERK',20000);
.
.
.
ERROR at line 1:
ORA-02290: check constraint (HR.MAX_EMP_SAL) violated
```

A single column can have multiple check constraints that reference the column in its definition. For example, the `salary` column could have one constraint that prevents values over 10000 and a separate constraint that prevents values less than 500.

If multiple check constraints exist for a column, then they must be designed so their purposes do not conflict. No order of evaluation of the conditions can be assumed. The database does not verify that check conditions are not mutually exclusive.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about restrictions for check constraints

# States of Integrity Constraints

As part of constraint definition, you can specify how and when Oracle Database should enforce the constraint, thereby determining the constraint state.

- Checks for Modified and Existing Data
  The database enables you to specify whether a constraint applies to existing data or future data. If a constraint is enabled, then the database checks new data as it is entered or updated. Data that does not conform to the constraint cannot enter the database.
- When the Database Checks Constraints for Validity
  Every constraint is either in a not deferrable (default) or deferrable state. This state determines when Oracle Database checks the constraint for validity.
- Examples of Constraint Checking
  The following examples help illustrate when Oracle Database performs the checking of constraints.

## Checks for Modified and Existing Data

The database enables you to specify whether a constraint applies to existing data or future data. If a constraint is enabled, then the database checks new data as it is entered or updated. Data that does not conform to the constraint cannot enter the database.

For example, enabling a NOT NULL constraint on employees.department_id guarantees that every future row has a department ID. If a constraint is disabled, then the table can contain rows that violate the constraint.

You can set constraints to either of the following validation modes:

- VALIDATE

  Existing data must conform to the constraint. For example, enabling a NOT NULL constraint on employees.department_id and setting it to VALIDATE checks that every existing row has a department ID.

- NOVALIDATE

  Existing data need not conform to the constraint. In effect, this is a “trust me” mode. For example, if you are certain that every sale that you loaded into a table has a date, then you can create a NOT NULL constraint on the date column and set the constraint to NOVALIDATE. Unenforced constraints are typically useful only with materialized views and query rewrite.

  For a constraint in NOVALIDATE mode, the RELY parameter indicates that the optimizer can use the constraint to determine join information. Even though the constraint is not used for validating data, it enables more sophisticated query rewrites for materialized views, and enables data warehousing tools to retrieve constraint information from the data dictionary. The default is NORELY, which means that the optimizer is effectively unaware of the constraint.

The behavior of VALIDATE and NOVALIDATE always depends on whether the constraint is enabled or disabled. The following table summarizes the relationships.

**Table 7-4 Checks on Modified and Existing Data**

| Modified Data | Existing Data | Summary |
|---|---|---|
| ENABLE | VALIDATE | Existing and future data must obey the constraint. An attempt to apply a new constraint to a populated table results in an error if existing rows violate the constraint. |
| ENABLE | NOVALIDATE | The database checks the constraint, but it need not be true for all rows. Therefore, existing rows can violate the constraint, but new or modified rows must conform to the rules. This mode is often used in data warehouses that contain existing data whose integrity has already been verified. |
| DISABLE | VALIDATE | The database disables the constraint, drops its index, and prevents modification of the constrained columns. |
| DISABLE | NOVALIDATE | The constraint is not checked and is not necessarily true. |

**See Also:**

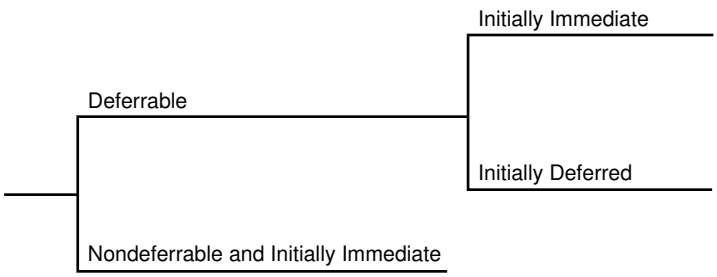
*Oracle Database SQL Language Reference* to learn about constraint states

## When the Database Checks Constraints for Validity

Every constraint is either in a not deferrable (default) or deferrable state. This state determines when Oracle Database checks the constraint for validity.

The following graphic shows the options for deferrable constraints.

**Figure 7-3 Options for Deferrable Constraints**




- Nondeferrable Constraints
  In a **nondeferrable constraint**, Oracle Database never defers the validity check of the constraint to the end of the transaction. Instead, the database checks the constraint at the end of each statement. If the constraint is violated, then the statement rolls back.
- Deferrable Constraints
  A **deferrable constraint** permits a transaction to use the `SET CONSTRAINT` clause to defer checking of this constraint until a `COMMIT` statement is issued. If you make changes to the database that might violate the constraint, then this setting effectively enables you to disable the constraint until all changes are complete.

## Nondeferrable Constraints

In a **nondeferrable constraint**, Oracle Database never defers the validity check of the constraint to the end of the transaction. Instead, the database checks the constraint at the end of each statement. If the constraint is violated, then the statement rolls back.

For example, a nondeferrable `NOT NULL` constraint exists for the `employees.last_name` column. If a session attempts to insert a row with no last name, then the database immediately rolls back the statement because the `NOT NULL` constraint is violated. No row is inserted.

## Deferrable Constraints

A **deferrable constraint** permits a transaction to use the `SET CONSTRAINT` clause to defer checking of this constraint until a `COMMIT` statement is issued. If you make changes to the database that might violate the constraint, then this setting effectively enables you to disable the constraint until all changes are complete.

You can set the default behavior for when the database checks the deferrable constraint. You can specify either of the following attributes:

- `INITIALLY IMMEDIATE`

  The database checks the constraint immediately after each statement executes. If the constraint is violated, then the database rolls back the statement.

- `INITIALLY DEFERRED`

  The database checks the constraint when a `COMMIT` is issued. If the constraint is violated, then the database rolls back the transaction.

Assume that a deferrable `NOT NULL` constraint on `employees.last_name` is set to `INITIALLY DEFERRED`. A user creates a transaction with 100 `INSERT` statements, some of which have null

values for `last_name`. When the user attempts to commit, the database rolls back all 100 statements. However, if this constraint were set to `INITIALLY IMMEDIATE`, then the database would not roll back the transaction.

If a constraint causes an action, then the database considers this action as part of the statement that caused it, whether the constraint is deferred or immediate. For example, deleting a row in `departments` causes the deletion of all rows in `employees` that reference the deleted department row. In this case, the deletion from `employees` is considered part of the `DELETE` statement executed against `departments`.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about constraint attributes and their default values

## Examples of Constraint Checking

The following examples help illustrate when Oracle Database performs the checking of constraints.

Assume the following:

- The `employees` table has the structure shown in "Self-Referential Integrity Constraints".
- The self-referential constraint makes entries in the `manager_id` column dependent on the values of the `employee_id` column.
- Example: Insertion of a Value in a Foreign Key Column When No Parent Key Value Exists
  This example concerns the insertion of the first row into the `employees` table. No rows currently exist, so how can a row be entered if the value in the `manager_id` column cannot reference an existing value in the `employee_id` column?
- Example: Update of All Foreign Key and Parent Key Values
  In this example, a self-referential constraint makes entries in the `manager_id` column of `employees` dependent on the values of the `employee_id` column.

### Example: Insertion of a Value in a Foreign Key Column When No Parent Key Value Exists

This example concerns the insertion of the first row into the `employees` table. No rows currently exist, so how can a row be entered if the value in the `manager_id` column cannot reference an existing value in the `employee_id` column?

Some possibilities are:

- If the `manager_id` column does not have a `NOT NULL` constraint defined on it, then you can enter a null for the `manager_id` column of the first row.

  Because nulls are allowed in foreign keys, Oracle Database inserts this row into the table.
- You can enter the same value in the `employee_id` and `manager_id` columns, specifying that the employee is his or her own manager.

  This case reveals that Oracle Database performs its constraint checking *after* the statement executes. To allow a row to be entered with the same values in the parent key

and the foreign key, the database must first insert the new row, and then determine whether any row in the table has an `employee_id` that corresponds to the `manager_id` of the new row.

- A multiple row `INSERT` statement, such as an `INSERT` statement with nested `SELECT` statements, can insert rows that reference one another.

  For example, the first row might have 200 for employee ID and 300 for manager ID, while the second row has 300 for employee ID and 200 for manager. Constraint checking is deferred until the complete execution of the `INSERT` statement. The database inserts all rows, and then checks all rows for constraint violations.

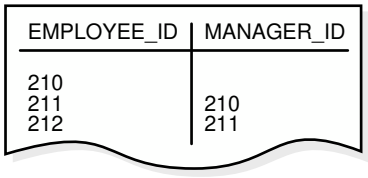
Default values are included as part of an `INSERT` statement before the statement is parsed. Thus, default column values are subject to all integrity constraint checking.

## Example: Update of All Foreign Key and Parent Key Values

In this example, a self-referential constraint makes entries in the `manager_id` column of `employees` dependent on the values of the `employee_id` column.

The company has been sold. Because of this sale, all employee numbers must be updated to be the current value plus 5000 to coordinate with the employee numbers of the new company. As shown in the following graphic, some employees are also managers:

**Figure 7-4 The employees Table Before Updates**

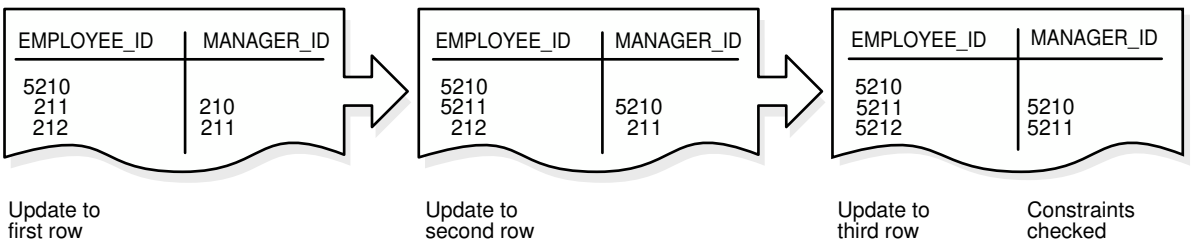



Because manager numbers are also employee numbers, the manager numbers must also increase by 5000. You could execute the following SQL statement to update the values:

```
UPDATE employees SET employee_id = employee_id + 5000,
  manager_id = manager_id + 5000;
```

Although a constraint is defined to verify that each `manager_id` value matches an `employee_id` value, the preceding statement is valid because the database effectively checks constraints after the statement completes. Figure 7-5 shows that the database performs the actions of the entire SQL statement before checking constraints.

**Figure 7-5 Constraint Checking**



The examples in this section illustrate the constraint checking mechanism during `INSERT` and `UPDATE` statements, but the database uses the same mechanism for all types of DML statements. The database uses the same mechanism for all types of constraints, not just self-referential constraints.

> **Note:**
>
> Operations on a view or synonym are subject to the integrity constraints defined on the base tables.

8

# Data Dictionary and Dynamic Performance Views

The central set of read-only reference tables and views of each Oracle database is known collectively as the **data dictionary**. The **dynamic performance views** are special views that are continuously updated while a database is open and in use.

This chapter contains the following sections:


- Overview of the Data Dictionary
- Overview of the Dynamic Performance Views
- Database Object Metadata


- Overview of the Data Dictionary
  An important part of an Oracle database is its data dictionary, which is a read-only set of tables that provides administrative metadata about the database.
- Overview of the Dynamic Performance Views
  Throughout its operation, Oracle Database maintains a set of virtual tables that record current database activity.
- Database Object Metadata
  The `DBMS_METADATA` package provides interfaces for extracting complete definitions of database objects.

## Overview of the Data Dictionary

An important part of an Oracle database is its data dictionary, which is a read-only set of tables that provides administrative metadata about the database.

- Purpose of the Data Dictionary
  The data dictionary contains metadata describing the contents of the database.
- Data Dictionary Components
  The data dictionary consists of base tables and views.
- How the Data Dictionary Works
  The Oracle Database user account `SYS` owns all base tables and user-accessible views of the data dictionary.
- Data Dictionary Storage
  The data dictionary that stores the metadata for the CDB as a whole is stored only in the system tablespaces.

### Purpose of the Data Dictionary

The data dictionary contains metadata describing the contents of the database.

For example, the data dictionary contains information such as the following:

- The definitions of every schema object in the database, including default values for columns and integrity constraint information

- The amount of space allocated for and currently used by the schema objects
- The names of Oracle Database users, privileges and roles granted to users, and auditing information related to users
- Data Management
  The data dictionary is a central part of data management for every Oracle database.
- Data Dictionary Separation in a CDB
  In a CDB, the data dictionary metadata is split between the CDB root and the PDBs. From the user and application perspective, the data dictionary in each container in a CDB is separate.

## Data Management

The data dictionary is a central part of data management for every Oracle database.

For example, the database performs the following actions:

- Accesses the data dictionary to find information about users, schema objects, and storage structures
- Modifies the data dictionary every time that a DDL statement is issued

Because Oracle Database stores data dictionary data in tables, just like other data, users can query the data with SQL. For example, users can run `SELECT` statements to determine their privileges, which tables exist in their schema, which columns are in these tables, whether indexes are built on these columns, and so on.

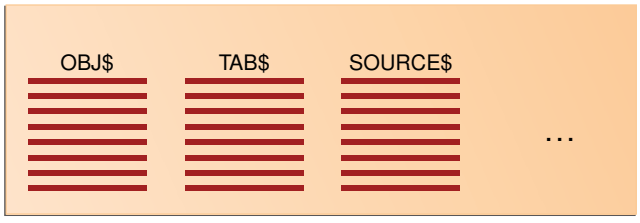
> **See Also:**
>
> - "Introduction to Schema Objects"
> - *Oracle Database Security Guide* to learn about user accounts
> - "Data Definition Language (DDL) Statements"

## Data Dictionary Separation in a CDB

In a CDB, the data dictionary metadata is split between the CDB root and the PDBs. From the user and application perspective, the data dictionary in each container in a CDB is separate.

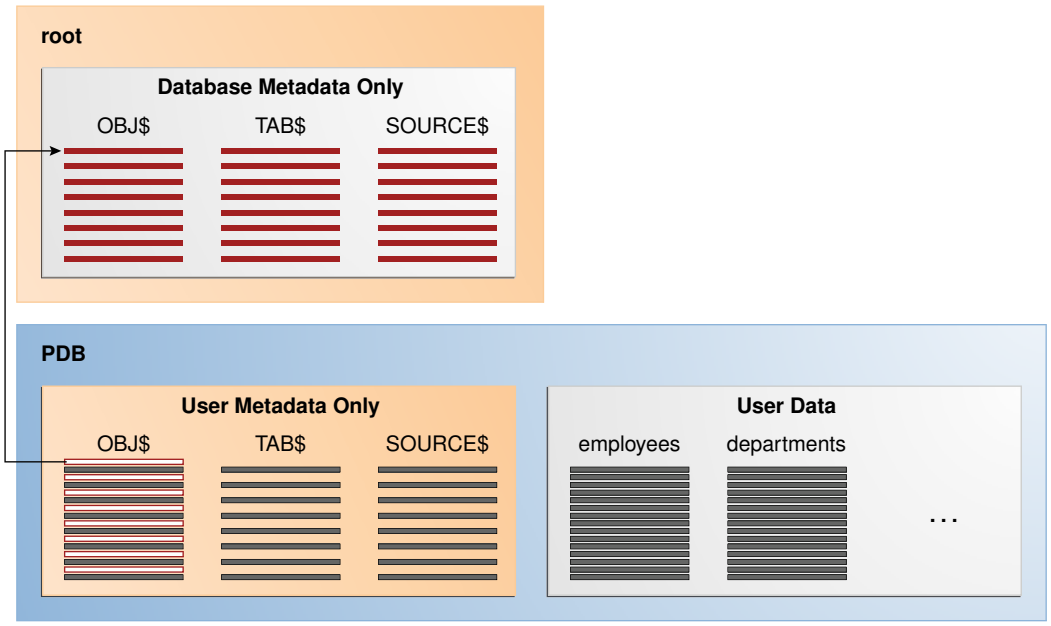
In a newly created CDB that does not yet contain user data, the data dictionary in the CDB root contains only system metadata. For example, the `TAB$` table contains rows that describe only Oracle-supplied tables, for example, `TRIGGER$` and `SERVICE$`. The following graphic depicts three underlying data dictionary tables, with the red bars indicating rows describing the system.

**Figure 8-1 Data Dictionary Metadata in the CDB Root**



Assume that you create a PDB, and then create an `hr` schema containing the `employees` and `departments` tables in this PDB. The data dictionary in the PDB contains some rows that describe Oracle-supplied entities, and other rows that describe user-created entities. For example, the `TAB$` table in the PDB dictionary has a row of metadata for the `employees` table and a row for the `departments` table.

**Figure 8-2 Data Dictionary Architecture in a CDB**




The preceding graphic shows that the data dictionary in the PDB contains *pointers* to the data dictionary in the CDB root. Internally, Oracle-supplied objects such as data dictionary table definitions and PL/SQL packages are represented *only once* in the CDB root. This architecture achieves two main goals within the CDB:

- Reduction of duplication

  For example, instead of storing the source code for the `DBMS_ADVISOR` PL/SQL package in every PDB, the CDB stores the code only once in `CDB$ROOT`, which saves disk space.

- Ease of database upgrade

  If the definition of a data dictionary table existed in every PDB, and if the definition were to change in a new release, then each PDB would need to be upgraded separately to capture the change. Storing the table definition only once in the CDB root eliminates this problem.

# Data Dictionary Components

The data dictionary consists of base tables and views.

These objects are defined as follows:

- Base tables

  These store information about the database. Only Oracle Database should write to and read these tables. Users rarely access the base tables directly because they are normalized and most data is stored in a cryptic format.

- Views

  These decode the base table data into useful information, such as user or table names, using joins and `WHERE` clauses to simplify the information. The views contain the names and description of all objects in the data dictionary. Some views are accessible to all database users, whereas others are intended for administrators only.

Typically, data dictionary views are grouped in sets. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes, as shown in the following table. By querying the appropriate views, you can access only the information relevant for you.

**Table 8-1 Data Dictionary View Sets**

| Prefix | User Access | Contents | Notes |
|---|---|---|---|
| `DBA_` | Database administrators | All objects | Some `DBA_` views have additional columns containing information useful to the administrator. |
| `ALL_` | All users | Objects to which user has privileges | Includes objects owned by user. These views obey the current set of enabled roles. |
| `USER_` | All users | Objects owned by user | Views with the prefix `USER_` usually exclude the column `OWNER`. This column is implied in the `USER_` views to be the user issuing the query. |

Not all views sets have three members. For example, the data dictionary contains a `DBA_LOCK` view but no `ALL_LOCK` view.

The system-supplied `DICTIONARY` view contains the names and abbreviated descriptions of all data dictionary views. The following query of this view includes partial sample output:

```
SQL> SELECT * FROM DICTIONARY
  2  ORDER BY TABLE_NAME;

TABLE_NAME                     COMMENTS
------------------------------ ----------------------------------------
ALL_ALL_TABLES                 Description of all object and relational
                               tables accessible to the user

ALL_APPLY                      Details about each apply process that
                               dequeues from the queue visible to the
                               current user
.
.
.
```

- Container Data Objects in a CDB
  A **container data object** is a table or view containing data pertaining to multiple containers or the whole CDB.
- Views with the Prefix DBA_
  Views with the prefix `DBA_` show all relevant information in the entire database. `DBA_` views are intended only for administrators.

- Views with the Prefix ALL_
  Views with the prefix `ALL_` refer to the user's overall perspective of the database.
- Views with the Prefix USER_
  The views most likely to be of interest to typical database users are those with the prefix `USER_`.
- The DUAL Table
  `DUAL` is a small table in the data dictionary that Oracle Database and user-written programs can reference to guarantee a known result.

> **See Also:**
>
> - "Overview of Views"
> - *Oracle Database Reference* for a complete list of data dictionary views and their columns

## Container Data Objects in a CDB

A **container data object** is a table or view containing data pertaining to multiple containers or the whole CDB.

Container data privileges support a general requirement in which multiple PDBs reside in a single CDB, but with different local administration requirements. For example, if application DBAs do not want to administer locally, then they can grant container data privileges on appropriate views to the common users. In this case, the CDB administrator can access the data for these PDBs. In contrast, PDB administrators who do not want the CDB administrator accessing their data do not grant container data privileges.

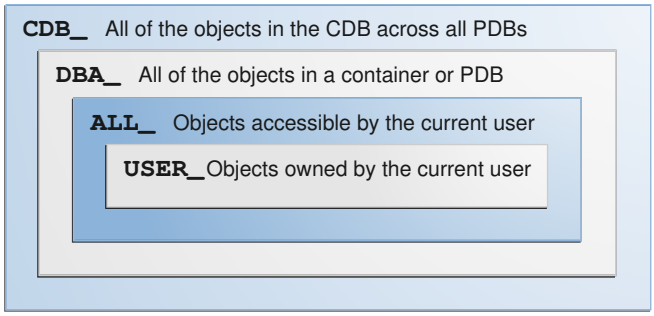
Examples of container data objects are Oracle-supplied views whose names begin with `V$` and `CDB_`. All container data objects have a `CON_ID` column. The following table shows the meaning of the values for this column.

**Table 8-2 Container ID Values**

| Container ID | Rows pertain to |
|---|---|
| 0 | Whole CDB |
| 1 | CDB$ROOT |
| 2 | PDB$SEED |
| All Other IDs | User-created PDBs, application roots, or application seeds |

In a CDB, for every `DBA_` view, a corresponding `CDB_` view exists. The owner of a `CDB_` view is the owner of the corresponding `DBA_` view. The following graphic shows the relationship among the different categories of dictionary views:

**Figure 8-3 Dictionary Views in a CDB**




When the current container is a PDB, a user can view data dictionary information for the current PDB only. When the current container is the CDB root, however, a common user can query `CDB_` views to see metadata for the CDB root and for PDBs for which this user is privileged.

> **Note:**
>
> When queried from the CDB root, `CDB_` and `V$` views implicitly convert data to the AL32UTF8 character set. If a character set needs more bytes to represent a character when converted to AL32UTF8, and if the view column width cannot accommodate data from a specific PDB, then data truncation is possible.

The following table describes queries of `CDB_` views. Each row describes an action that occurs after the action in the preceding row.

**Table 8-3 Querying CDB_ Views**

| Operation | Description |
|---|---|
| `SQL> CONNECT SYSTEM`<br>`Enter password: ********`<br>`Connected.` | The `SYSTEM` user, which is common to all containers in the CDB, connects to the CDB root (see "Common User Accounts"). |
| `SQL> SELECT COUNT(*) FROM CDB_USERS`<br>`WHERE CON_ID=1;`<br><br>`COUNT(*)`<br>`--------`<br>`38` | `SYSTEM` queries `CDB_USERS` to obtain the number of common users in the CDB root. The output indicates that 38 common users exist in CDB root. |

**Table 8-3 (Cont.) Querying CDB_ Views**

| Operation | Description |
|---|---|
| `SQL> SELECT COUNT(DISTINCT(CON_ID)) FROM CDB_USERS;`<br><br>`COUNT(DISTINCT(CON_ID))`<br>`----------`<br>`4` | SYSTEM queries CDB_USERS to determine the number of distinct containers in the CDB. |
| `SQL> CONNECT SYSTEM@hrdb`<br>`Enter password: ********`<br>`Connected.` | The SYSTEM user now connects to the PDB named hrpdb. |
| `SQL> SELECT COUNT(*) FROM CDB_USERS;`<br><br>`COUNT(*)`<br>`----------`<br>`48` | SYSTEM queries CDB_USERS. The output indicates that 48 common and local users exist in the current container, which is hrpdb. |
| `SQL> SELECT COUNT(*) FROM DBA_USERS;`<br><br>`COUNT(*)`<br>`----------`<br>`48` | SYSTEM queries DBA_USERS. The output is the same as the previous query. Because SYSTEM is *not* connected to the CDB root, the DBA_USERS view shows the same output as CDB_USERS. Because DBA_USERS only shows the users in the *current* container, it shows 48. |

## Views with the Prefix DBA_

Views with the prefix DBA_ show all relevant information in the entire database. DBA_ views are intended only for administrators.

The following sample query shows information about all objects in the database:

```
SELECT OWNER, OBJECT_NAME, OBJECT_TYPE
FROM   DBA_OBJECTS
ORDER BY OWNER, OBJECT_NAME;
```

> **See Also:**
>
> *Oracle Database Administrator's Guide* for detailed information on administrative privileges

## Views with the Prefix ALL_

Views with the prefix `ALL_` refer to the user's overall perspective of the database.

These views return information about schema objects to which the user has access through public or explicit grants of privileges and roles, in addition to schema objects that the user owns.

For example, the following query returns information about all the objects to which you have access:

```
SELECT OWNER, OBJECT_NAME, OBJECT_TYPE
FROM   ALL_OBJECTS
ORDER BY OWNER, OBJECT_NAME;
```

Because the `ALL_` views obey the current set of enabled roles, query results depend on which roles are enabled, as shown in the following example:

```
SQL> SET ROLE ALL;

Role set.

SQL> SELECT COUNT(*) FROM ALL_OBJECTS;

COUNT(*)
----------
68295

SQL> SET ROLE NONE;

Role set.

SQL> SELECT COUNT(*) FROM ALL_OBJECTS;

COUNT(*)
----------
53771
```

Application developers should be cognizant of the effect of roles when using `ALL_` views in a stored procedure, where roles are not enabled by default.

## Views with the Prefix USER_

The views most likely to be of interest to typical database users are those with the prefix `USER_`.

These views:

- Refer to the user's private environment in the database, including metadata about schema objects created by the user, grants made by the user, and so on
- Display only rows pertinent to the user, returning a subset of the information in the `ALL_` views
- Has columns identical to the other views, except that the column `OWNER` is implied
- Can have abbreviated `PUBLIC` synonyms for convenience

For example, the following query returns all the objects contained in your schema:

```
SELECT OBJECT_NAME, OBJECT_TYPE
FROM   USER_OBJECTS
ORDER BY OBJECT_NAME;
```

## The DUAL Table

`DUAL` is a small table in the data dictionary that Oracle Database and user-written programs can reference to guarantee a known result.

The dual table is useful when a value must be returned only once, for example, the current date and time. All database users have access to `DUAL`.

The `DUAL` table has one column called `DUMMY` and one row containing the value `X`. The following example queries `DUAL` to perform an arithmetical operation:

```
SQL> SELECT ((3*4)+5)/3 FROM DUAL;

((3*4)+5)/3
-----------
 5.66666667
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information about the `DUAL` table

# How the Data Dictionary Works

The Oracle Database user account `SYS` owns all base tables and user-accessible views of the data dictionary.

During database operation, Oracle Database reads the data dictionary to ascertain that schema objects exist and that users have proper access to them. Oracle Database updates the data dictionary continuously to reflect changes in database structures, auditing, grants, and data.

For example, if user `hr` creates a table named `interns`, then the database adds new rows to the data dictionary that reflect the new table, columns, segment, extents, and the privileges that `hr` has on the table. This new information is visible the next time the dictionary views are queried.

Data in the base tables of the data dictionary *is necessary for Oracle Database to function*. Only Oracle Database should write or change data dictionary information. No Oracle Database user should *ever* alter rows or schema objects contained in the `SYS` schema because such activity can compromise data integrity. The security administrator must keep strict control of this central account.

> **WARNING:**
>
> Altering or manipulating the data in data dictionary tables can permanently and detrimentally affect database operation.

- Metadata and Data Links
  The CDB uses an internal linking mechanism to separate data dictionary information.
- Public Synonyms for Data Dictionary Views
  Oracle Database creates public **synonyms** for many data dictionary views so users can access them conveniently.
- Data Dictionary Cache
  Much of the data dictionary information is in the **data dictionary cache** because the database constantly requires the information to validate user access and verify the state of schema objects.
- Other Programs and the Data Dictionary
  Other Oracle Database products can reference existing views and create additional data dictionary tables or views of their own.

> **See Also:**
>
> "SYS and SYSTEM Accounts"

## Metadata and Data Links

The CDB uses an internal linking mechanism to separate data dictionary information.

Specifically, Oracle Database uses the following automatically managed pointers:

- Metadata links

  Oracle Database stores metadata about dictionary objects only in the CDB root. For example, the column definitions for the `OBJ$` dictionary table, which underlies the `DBA_OBJECTS` data dictionary view, exist only in the CDB root. As depicted in Figure 8-2, the `OBJ$` table in each PDB uses an internal mechanism called a metadata link to point to the definition of `OBJ$` stored in the CDB root.

  The *data* corresponding to a metadata link resides in its PDB, not in the CDB root. For example, if you create table `mytable` in `hrpdb` and add rows to it, then the rows are stored in the PDB data files, not in the CDB root data files.

  The data dictionary views in the PDB and in the CDB root contain different rows. For example, a new row describing `mytable` exists in the `OBJ$` table in `hrpdb`, but not in the `OBJ$` table in the CDB root. Thus, a query of `DBA_OBJECTS` in the CDB root and `DBA_OBJECTS` in `hrdpb` shows different results.
- Data links

> **Note:**
>
> Data links were called *object links* in Oracle Database 12c Release 1 (12.1.0.2).

In some cases, Oracle Database stores the data (not only metadata) for an object only once in the application root of an application container. Consider an e-commerce company that has different PDBs for different regions. The application root might store a table named `postal_codes`, which lists all U.S. zip codes. Every application PDB in this container requires access to the common `postal_codes` table.

An application PDB uses an internal mechanism called a data link to refer to the object in the application root. The application PDB in which the data link was created also stores the data link description. A data link inherits the data type of the object to which it refers.

- Extended data link

  An extended data link is a hybrid of a data link and a metadata link. Like a data link, an extended data link refers to an object in an application root. However, the extended data link also refers to a corresponding object in the application PDB. For example, an application PDB might have an extended data link table that stores both U.S. zip codes and Canadian postal codes. Like a metadata link, the object in the application PDB inherits metadata from the corresponding object in the application root.

  When queried in the application root, an extended data-linked table fetches rows only from the application root, for example, just U.S. zip codes. However, when queried in an application PDB, an extended data-linked table fetches rows from both the application root and application PDB, for example, U.S. zip codes and Canadian postal codes.

Oracle Database automatically creates and manages metadata and data links to `CDB$ROOT`. Users cannot add, modify, or remove these links.

> **See Also:**
>
> - "Application Common Objects"
> - *Oracle Database Concepts* for an overview of the data dictionary

## Public Synonyms for Data Dictionary Views

Oracle Database creates public **synonyms** for many data dictionary views so users can access them conveniently.

The security administrator can also create additional public synonyms for schema objects that are used systemwide. Oracle recommends against using the same name for a private schema object and a public synonym.

> **See Also:**
>
> "Overview of Synonyms"

## Data Dictionary Cache

Much of the data dictionary information is in the **data dictionary cache** because the database constantly requires the information to validate user access and verify the state of schema objects.

The caches typically contain the parsing information. The `COMMENTS` columns describing the tables and their columns are not cached in the dictionary cache, but may be cached in the **database buffer cache**.

> **See Also:**
>
> "Data Dictionary Cache"

## Other Programs and the Data Dictionary

Other Oracle Database products can reference existing views and create additional data dictionary tables or views of their own.

Oracle recommends that application developers who write programs referring to the data dictionary use the public synonyms rather than the underlying tables. Synonyms are less likely to change between releases.

## Data Dictionary Storage

The data dictionary that stores the metadata for the CDB as a whole is stored only in the system tablespaces.

The data dictionary that stores the metadata for a specific PDB is stored in the self-contained tablespaces dedicated to this PDB. The PDB tablespaces contain both the data and metadata for an application back end. Thus, each set of data dictionary tables is stored in its own dedicated set of tablespaces.

> **See Also:**
>
> "The SYSTEM Tablespace" for more information about the `SYSTEM` tablespace

# Overview of the Dynamic Performance Views

Throughout its operation, Oracle Database maintains a set of virtual tables that record current database activity.

These views are dynamic because they are continuously updated while a database is open and in use. The views are sometimes called *V$ views* because their names begin with `V$`.

Dynamic performance views contain information such as the following:

- System and session parameters

- Memory usage and allocation
- File states (including RMAN backup files)
- Progress of jobs and tasks
- SQL execution
- Statistics and metrics

The dynamic performance views have the following primary uses:

- Oracle Enterprise Manager uses the views to obtain information about the database.
- Administrators can use the views for performance monitoring and debugging.

- Contents of the Dynamic Performance Views
  Dynamic performance views are called *fixed views* because they cannot be altered or removed by a database administrator. However, database administrators can query and create views on the tables and grant access to these views to other users.
- Storage of the Dynamic Performance Views
  Dynamic performance views are based on virtual tables built from database memory structures.

> **See Also:**
>
> *Oracle Database Reference* for a complete list of the dynamic performance views

## Contents of the Dynamic Performance Views

Dynamic performance views are called *fixed views* because they cannot be altered or removed by a database administrator. However, database administrators can query and create views on the tables and grant access to these views to other users.

`SYS` owns the dynamic performance tables, whose names begin with `V_$`. Views are created on these tables, and then public synonyms prefixed with `V$`. For example, the `V$DATAFILE` view contains information about data files. The `V$FIXED_TABLE` view contains information about all of the dynamic performance tables and views.

For almost every `V$` view, a corresponding `GV$` view exists. In Oracle Real Application Clusters (Oracle RAC), querying a `GV$` view retrieves the `V$` view information from all qualified database instances.

When you use the Database Configuration Assistant (DBCA) to create a database, Oracle automatically creates the data dictionary. Oracle Database automatically runs the `catalog.sql` script, which contains definitions of the views and public synonyms for the dynamic performance views. You must run `catalog.sql` to create these views and synonyms.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to run `catalog.sql` manually
> - *Oracle Real Application Clusters Administration and Deployment Guide* to learn about using performance views in Oracle RAC

## Storage of the Dynamic Performance Views

Dynamic performance views are based on virtual tables built from database memory structures.

The views are not conventional tables stored in the database. Read consistency is not guaranteed for the views because the data is updated dynamically.

Because the dynamic performance views are not true tables, the data depends on the state of the database and database instance. For example, you can query `V$INSTANCE` and `V$BGPROCESS` when the database is started but not mounted. However, you cannot query `V$DATAFILE` until the database has been mounted.

> **See Also:**
>
> "Data Concurrency and Consistency "

# Database Object Metadata

The `DBMS_METADATA` package provides interfaces for extracting complete definitions of database objects.

The definitions can be expressed either as XML or as SQL DDL. Oracle Database provides two styles of interface: a flexible, sophisticated interface for programmatic control, and a simplified interface for ad hoc querying.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_METADATA`

# Part III
# Oracle Data Access

**Structured Query Language (SQL)** is the high-level declarative computer language with which all programs and users access data in an Oracle database. **PL/SQL** and **Java**, which are server-side procedural languages, enable you to store data logic in the database itself.

This part contains the following chapters:

- SQL
- Server-Side Programming: PL/SQL and Java

- SQL
  This chapter provides an overview of the **Structured Query Language (SQL)** and how Oracle Database processes SQL statements.
- Server-Side Programming: PL/SQL and Java
  **SQL** explains the Structured Query Language (SQL) language and how the database processes SQL statements. This chapter explains how Procedural Language/SQL (PL/SQL) or Java programs stored in the database can use SQL.

# 9
# SQL

This chapter provides an overview of the **Structured Query Language (SQL)** and how Oracle Database processes SQL statements.

This chapter includes the following topics:


- Introduction to SQL
- Overview of SQL Statements
- Overview of the Optimizer
- Overview of SQL Processing


- Introduction to SQL
  SQL (pronounced *sequel*) is the set-based, high-level declarative computer language with which all programs and users access data in an Oracle database.
- Overview of SQL Statements
  All operations performed on the information in an Oracle database are run using SQL **statements**. A SQL statement is a computer program or instruction that consists of identifiers, parameters, variables, names, data types, and SQL **reserved words**.
- Overview of the Optimizer
  To understand how Oracle Database processes SQL statements, it is necessary to understand the part of the database called the **optimizer** (also known as the *query optimizer* or *cost-based optimizer*). All SQL statements use the optimizer to determine the most efficient means of accessing the specified data.
- Overview of SQL Processing
  This section explains how Oracle Database processes SQL statements. Specifically, the section explains the way in which the database processes DDL statements to create objects, DML to modify data, and queries to retrieve data.

## Introduction to SQL

SQL (pronounced *sequel*) is the set-based, high-level declarative computer language with which all programs and users access data in an Oracle database.

Although some Oracle tools and applications mask SQL use, all database tasks are performed using SQL. Any other data access method circumvents the security built into Oracle Database and potentially compromises data security and integrity.

SQL provides an interface to a relational database such as Oracle Database. SQL unifies tasks such as the following in one consistent language:

- Creating, replacing, altering, and dropping objects
- Inserting, updating, and deleting table rows
- Querying data
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL can be used interactively, which means that statements are entered manually into a program. SQL statements can also be embedded within a program written in a different language such as C or Java.

- SQL Data Access
  There are two broad families of computer languages: **declarative languages** that are nonprocedural and describe *what* should be done, and **procedural languages** such as C++ and Java that describe *how* things should be done.
- SQL Standards
  Oracle strives to follow industry-accepted standards and participates actively in SQL standards committees.

> **See Also:**
>
> - "Introduction to Server-Side Programming"
> - *Oracle Database Development Guide* to learn how to choose a programming environment
> - *Oracle Database SQL Language Reference* for an introduction to SQL

## SQL Data Access

There are two broad families of computer languages: **declarative languages** that are nonprocedural and describe *what* should be done, and **procedural languages** such as C++ and Java that describe *how* things should be done.

SQL is declarative in the sense that users specify the result that they want, not how to derive it. For example, the following statement queries records for employees whose last name begins with `K`:

The database performs the work of generating a procedure to navigate the data and retrieve the requested results. The declarative nature of SQL enables you to work with data at the logical level. You need be concerned with implementation details only when you manipulate the data.

```
SELECT   last_name, first_name
FROM     hr.employees
WHERE    last_name LIKE 'K%'
ORDER BY last_name, first_name;
```

The database retrieves all rows satisfying the `WHERE` condition, also called the predicate, in a single step. The database can pass these rows as a unit to the user, to another SQL statement, or to an application. The application does not need to process the rows one by one, nor does the developer need to know how the rows are physically stored or retrieved.

All SQL statements use the optimizer, a component of the database that determines the most efficient means of accessing the requested data. Oracle Database also supports techniques that you can use to make the optimizer perform its job better.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for detailed information about SQL statements and other parts of SQL (such as operators, functions, and format models)

## SQL Standards

Oracle strives to follow industry-accepted standards and participates actively in SQL standards committees.

Industry-accepted committees are the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Both ANSI and the ISO/IEC have accepted SQL as the standard language for relational databases.

The SQL standard consists of ten parts. One part (SQL/RPR:2012) is new in 2102. Five other parts were revised in 2011. For the other four parts, the 2008 version remains in place.

Oracle SQL includes many extensions to the ANSI/ISO standard SQL language, and Oracle Database tools and applications provide additional statements. The tools SQL*Plus, SQL Developer, and Oracle Enterprise Manager enable you to run any ANSI/ISO standard SQL statement against an Oracle database and any additional statements or functions available for those tools.

> **See Also:**
>
> - *Oracle Database 2 Day Developer's Guide*
> - *Oracle Database SQL Language Reference* for an explanation of the differences between Oracle SQL and standard SQL
> - *SQL*Plus User's Guide and Reference* for SQL*Plus commands, including their distinction from SQL statements

# Overview of SQL Statements

All operations performed on the information in an Oracle database are run using SQL **statements**. A SQL statement is a computer program or instruction that consists of identifiers, parameters, variables, names, data types, and SQL **reserved words**.

> **Note:**
>
> SQL reserved words have special meaning in SQL and should not be used for any other purpose. For example, `SELECT` and `UPDATE` are reserved words and should not be used as table names.

A SQL statement must be the equivalent of a complete SQL sentence, such as:

```
SELECT last_name, department_id FROM employees
```

Oracle Database only runs complete SQL statements. A fragment such as the following generates an error indicating that more text is required:

```
SELECT last_name;
```

Oracle SQL statements are divided into the following categories:

- Data Definition Language (DDL) Statements
- Data Manipulation Language (DML) Statements
- Transaction Control Statements
- Session Control Statements
- System Control Statement
- Embedded SQL Statements

- Data Definition Language (DDL) Statements
  Data definition language (**DLL**) statements define, structurally change, and drop schema objects.
- Data Manipulation Language (DML) Statements
  Data manipulation language (**DML**) statements query or manipulate data in existing schema objects.
- Transaction Control Statements
  Transaction control statements manage the changes made by DML statements and group DML statements into transactions.
- Session Control Statements
  Session control statements dynamically manage the properties of a user **session**.
- System Control Statement
  A system control statement changes the properties of the **database instance**.
- Embedded SQL Statements
  Embedded SQL statements incorporate DDL, DML, and transaction control statements within a procedural language program.

# Data Definition Language (DDL) Statements

Data definition language (**DLL**) statements define, structurally change, and drop schema objects.

DDL enables you to alter attributes of an object without altering the applications that access the object. For example, you can add a column to a table accessed by a human resources application without rewriting the application. You can also use DDL to alter the structure of objects while database users are performing work in the database.

More specifically, DDL statements enable you to:

- Create, alter, and drop schema objects and other database structures, including the database itself and database users. Most DDL statements start with the keywords `CREATE`, `ALTER`, or `DROP`.
- Delete all the data in schema objects without removing the structure of these objects (`TRUNCATE`).

> **Note:**
>
> Unlike `DELETE`, `TRUNCATE` generates no undo data, which makes it faster than `DELETE`. Also, `TRUNCATE` does not invoke delete triggers

- Grant and revoke privileges and roles (`GRANT`, `REVOKE`).
- Turn auditing options on and off (`AUDIT`, `NOAUDIT`).
- Add a comment to the data dictionary (`COMMENT`).

**Example 9-1 DDL Statements**

The following example uses DDL statements to create the `plants` table and then uses DML to insert two rows in the table. The example then uses DDL to alter the table structure, grant and revoke read privileges on this table to a user, and then drop the table.

```
CREATE TABLE plants
    ( plant_id    NUMBER PRIMARY KEY,
      common_name VARCHAR2(15) );

INSERT INTO plants VALUES (1, 'African Violet'); # DML statement

INSERT INTO plants VALUES (2, 'Amaryllis'); # DML statement

ALTER TABLE plants ADD
    ( latin_name VARCHAR2(40) );

GRANT READ ON plants TO scott;

REVOKE READ ON plants FROM scott;

DROP TABLE plants;
```

An implicit `COMMIT` occurs immediately before the database executes a DDL statement and a `COMMIT` or `ROLLBACK` occurs immediately afterward. In the preceding example, two `INSERT` statements are followed by an `ALTER TABLE` statement, so the database commits the two `INSERT` statements. If the `ALTER TABLE` statement succeeds, then the database commits this statement; otherwise, the database rolls back this statement. In either case, the two `INSERT` statements have already been committed.

> **See Also:**
>
> - *Oracle Database Security Guide* to learn about privileges and roles
> - *Oracle Database 2 Day Developer's Guide* and *Oracle Database Administrator's Guide* to learn how to create schema objects
> - *Oracle Database Development Guide* to learn about the difference between blocking and nonblocking DDL
> - *Oracle Database SQL Language Reference* for a list of DDL statements

# Data Manipulation Language (DML) Statements

Data manipulation language (**DML**) statements query or manipulate data in existing schema objects.

Whereas DDL statements change the structure of the database, DML statements query or change the contents. For example, `ALTER TABLE` changes the structure of a table, whereas `INSERT` adds one or more rows to the table.

DML statements are the most frequently used SQL statements and enable you to:

- Retrieve or fetch data from one or more tables or views (`SELECT`).
- Add new rows of data into a table or view (`INSERT`) by specifying a list of column values or using a subquery to select and manipulate existing data.
- Change column values in existing rows of a table or view (`UPDATE`).
- Update or insert rows conditionally into a table or view (`MERGE`).
- Remove rows from tables or views (`DELETE`).
- View the execution plan for a SQL statement (`EXPLAIN PLAN`).
- Lock a table or view, temporarily limiting access by other users (`LOCK TABLE`).

The following example uses DML to query the `employees` table. The example uses DML to insert a row into `employees`, update this row, and then delete it:

```
SELECT * FROM employees;

INSERT INTO employees (employee_id, last_name, email, job_id, hire_date,
salary)
  VALUES (1234, 'Mascis', 'JMASCIS', 'IT_PROG', '14-FEB-2008', 9000);

UPDATE employees SET salary=9100 WHERE employee_id=1234;

DELETE FROM employees WHERE employee_id=1234;
```

A collection of DML statements that forms a logical unit of work is called a transaction. For example, a transaction to transfer money could involve three discrete operations: decreasing the savings account balance, increasing the checking account balance, and recording the transfer in an account history table. Unlike DDL statements, DML statements do not implicitly commit the current transaction.

- SELECT Statements
  A **query** is an operation that retrieves data from a table or view.
- Joins
  A **join** is a query that combines rows from two or more tables, views, or materialized views.
- Subqueries
  A **subquery** is a `SELECT` statement nested within another SQL statement. Subqueries are useful when you must execute multiple queries to solve a single problem.

> **See Also:**
>
> - "Differences Between DML and DDL Processing"
> - "Introduction to Transactions "
> - *Oracle Database 2 Day Developer's Guide* to learn how to query and manipulate data
> - *Oracle Database SQL Language Reference* for a list of DML statements

## SELECT Statements

A **query** is an operation that retrieves data from a table or view.

`SELECT` is the only SQL statement that you can use to query data. The set of data retrieved from execution of a `SELECT` statement is known as a **result set**.

The following table shows two required keywords and two keywords that are commonly found in a `SELECT` statement. The table also associates capabilities of a `SELECT` statement with the keywords.

**Table 9-1 Keywords in a SQL Statement**

| Keyword | Required? | Description | Capability |
|---|---|---|---|
| `SELECT` | Yes | Specifies which columns should be shown in the result. Projection produces a subset of the columns in the table.<br>An expression is a combination of one or more values, operators, and SQL functions that resolves to a value. The list of expressions that appears after the `SELECT` keyword and before the `FROM` clause is called the select list. | Projection |
| `FROM` | Yes | Specifies the tables or views from which the data should be retrieved. | Joining |
| `WHERE` | No | Specifies a condition to filter rows, producing a subset of the rows in the table. A condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`. | Selection |
| `ORDER BY` | No | Specifies the order in which the rows should be shown. | |

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `SELECT` syntax and semantics

## Joins

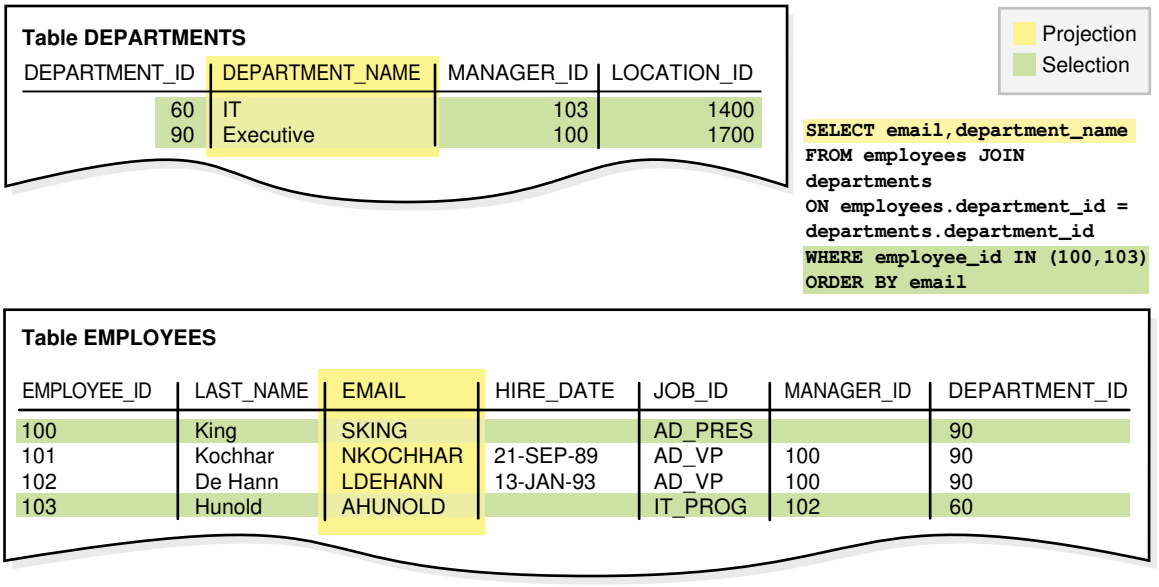
A **join** is a query that combines rows from two or more tables, views, or materialized views.

The following example joins the `employees` and `departments` tables (`FROM` clause), selects only rows that meet specified criteria (`WHERE` clause), and uses projection to retrieve data from two columns (`SELECT`). Sample output follows the SQL statement.

```
SELECT email, department_name
FROM   employees
JOIN   departments
ON     employees.department_id = departments.department_id
WHERE  employee_id IN (100,103)
ORDER BY email;

EMAIL                     DEPARTMENT_NAME
------------------------- ------------------------------
AHUNOLD                   IT
SKING                     Executive
```

The following graphic represents the operations of projection and selection in the join shown in the preceding query.

**Figure 9-1 Projection and Selection**




Most joins have at least one join condition, either in the `FROM` clause or in the `WHERE` clause, that compares two columns, each from a different table. The database combines pairs of rows, each containing one row from each table, for which the join condition evaluates to `TRUE`. The optimizer determines the order in which the database joins tables based on the join conditions, indexes, and any available statistics for the tables.

Join types include the following:

- Inner joins

  An inner join is a join of two or more tables that returns only rows that satisfy the join condition. For example, if the join condition is `employees.department_id=departments.department_id`, then rows that do not satisfy this condition are not returned.

- Outer joins

  An outer join returns all rows that satisfy the join condition and also returns rows from one table for which no rows from the other table satisfy the condition.

  The result of a left outer join for table *A* and *B* always contains all records of the left table *A*, even if the join condition does not match a record in the right table *B*. If no matching row from *B* exists, then *B* columns contain nulls for rows that have no match in *B*. For example, if not all employees are in departments, then a left outer join of `employees` (left table) and `departments` (right table) retrieves all rows in `employees` even if no rows in `departments` satisfy the join condition (`employees.department_id` is null).

  The result of a right outer join for table *A* and *B* contains all records of the right table *B*, even if the join condition does not match a row in the left table *A*. If no matching row from *A* exists, then *A* columns contain nulls for rows that have no match in *A*. For example, if not all departments have employees, a right outer join of `employees` (left table) and `departments` (right table) retrieves all rows in `departments` even if no rows in `employees` satisfy the join condition.

  A full outer join is the combination of a left outer join and a right outer join.

- Cartesian products

  If two tables in a join query have no join condition, then the database performs a Cartesian join. Each row of one table combines with each row of the other. For example, if `employees` has 107 rows and `departments` has 27, then the Cartesian product contains 107*27 rows. A Cartesian product is rarely useful.

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* to learn about joins
> - *Oracle Database SQL Language Reference* for detailed descriptions and examples of joins

## Subqueries

A **subquery** is a `SELECT` statement nested within another SQL statement. Subqueries are useful when you must execute multiple queries to solve a single problem.

Each query portion of a statement is called a query block. In the following query, the subquery in parentheses is the inner query block:

```
SELECT first_name, last_name
FROM   employees
WHERE  department_id
IN     ( SELECT department_id
         FROM departments
         WHERE location_id = 1800 );
```

The inner `SELECT` statement retrieves the IDs of departments with location ID 1800. These department IDs are needed by the outer query block, which retrieves names of employees in the departments whose IDs were supplied by the subquery.

The structure of the SQL statement does not force the database to execute the inner query first. For example, the database could rewrite the entire query as a join of `employees` and `departments`, so that the subquery never executes by itself. As another example, the Virtual Private Database (VPD) feature could restrict the query of employees using a `WHERE` clause, so that the database queries the employees first and then obtains the department IDs. The optimizer determines the best sequence of steps to retrieve the requested rows.

> **See Also:**
>
> *Oracle Database Security Guide* to learn more about VPD

## Transaction Control Statements

Transaction control statements manage the changes made by DML statements and group DML statements into transactions.

These statements enable you to:

- Make changes to a transaction permanent (`COMMIT`).
- Undo the changes in a transaction, since the transaction started (`ROLLBACK`) or since a savepoint (`ROLLBACK TO SAVEPOINT`). A savepoint is a user-declared intermediate marker within the context of a transaction.

  > **Note:**
  >
  > The `ROLLBACK` statement ends a transaction, but `ROLLBACK TO SAVEPOINT` does not.

- Set a point to which you can roll back (`SAVEPOINT`).
- Establish properties for a transaction (`SET TRANSACTION`).
- Specify whether a deferrable integrity constraint is checked following each DML statement or when the transaction is committed (`SET CONSTRAINT`).

The following example starts a transaction named `Update salaries`. The example creates a savepoint, updates an employee salary, and then rolls back the transaction to the savepoint. The example updates the salary to a different value and commits.

```
SET TRANSACTION NAME 'Update salaries';

SAVEPOINT before_salary_update;

UPDATE employees SET salary=9100 WHERE employee_id=1234 # DML

ROLLBACK TO SAVEPOINT before_salary_update;

UPDATE employees SET salary=9200 WHERE employee_id=1234 # DML

COMMIT COMMENT 'Updated salaries';
```

> **See Also:**
>
> - "Introduction to Transactions "
> - "When the Database Checks Constraints for Validity"
> - *Oracle Database SQL Language Reference* to learn about transaction control statements

## Session Control Statements

Session control statements dynamically manage the properties of a user **session**.

A 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 user disconnects or exits the database application.

Session control statements enable you to:

- Alter the current session by performing a specialized function, such as setting the default date format (`ALTER SESSION`).
- Enable and disable roles, which are groups of privileges, for the current session (`SET ROLE`).

The following statement dynamically changes the default date format for your session to `'YYYY MM DD-HH24:MI:SS'`:

```
ALTER SESSION
   SET NLS_DATE_FORMAT = 'YYYY MM DD HH24:MI:SS';
```

Session control statements do not implicitly commit the current transaction.

> **See Also:**
>
> - "Connections and Sessions"
> - *Oracle Database SQL Language Reference* for `ALTER SESSION` syntax and semantics

## System Control Statement

A system control statement changes the properties of the **database instance**.

The only system control statement is `ALTER SYSTEM`. It enables you to change settings such as the minimum number of shared servers, terminate a session, and perform other system-level tasks.

Examples of the system control statement include:

```
ALTER SYSTEM SWITCH LOGFILE;

ALTER SYSTEM KILL SESSION '39, 23';
```

The `ALTER SYSTEM` statement does not implicitly commit the current transaction.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `ALTER SYSTEM` syntax and semantics

## Embedded SQL Statements

Embedded SQL statements incorporate DDL, DML, and transaction control statements within a procedural language program.

Embedded statements are used with the Oracle precompilers. Embedded SQL is one approach to incorporating SQL in your procedural language applications. Another approach is to use a procedural API such as Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC).

Embedded SQL statements enable you to:

- Define, allocate, and release a cursor (`DECLARE CURSOR`, `OPEN`, `CLOSE`).
- Specify a database and connect to it (`DECLARE DATABASE`, `CONNECT`).
- Assign variable names (`DECLARE STATEMENT`).
- Initialize descriptors (`DESCRIBE`).
- Specify how error and warning conditions are handled (`WHENEVER`).
- Parse and run SQL statements (`PREPARE`, `EXECUTE`, `EXECUTE IMMEDIATE`).
- Retrieve data from the database (`FETCH`).

> **See Also:**
>
> - "Introduction to Server-Side Programming"
> - *Oracle Database Development Guide*

# Overview of the Optimizer

To understand how Oracle Database processes SQL statements, it is necessary to understand the part of the database called the **optimizer** (also known as the *query optimizer* or *cost-based optimizer*). All SQL statements use the optimizer to determine the most efficient means of accessing the specified data.

- Use of the Optimizer
  The optimizer generates execution plans describing possible methods of execution.
- Optimizer Components
  The optimizer contains three main components: the transformer, estimator, and plan generator.
- Access Paths
  An **access path** is the technique that a query uses to retrieve rows.
- Optimizer Statistics
  The **optimizer statistics** are a collection of data that describe details about the database and the objects in the database. The statistics provide a statistically correct picture of data storage and distribution usable by the optimizer when evaluating access paths.
- Optimizer Hints
  A **hint** is a comment in a SQL statement that acts as an instruction to the optimizer.

## Use of the Optimizer

The optimizer generates execution plans describing possible methods of execution.

The optimizer determines which execution plan is most efficient by considering several sources of information. For example, the optimizer considers query conditions, available access paths, statistics gathered for the system, and hints.

To execute a DML statement, Oracle Database may have to perform many steps. Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement. The steps that the database uses to execute a statement greatly affect how quickly the statement runs. Many different ways of processing a DML statement are often possible. For example, the order in which tables or indexes are accessed can vary.

When determining the best execution plan for a SQL statement, the optimizer performs the following operations:

- Evaluation of expressions and conditions
- Inspection of integrity constraints to learn more about the data and optimize based on this metadata
- Statement transformation
- Choice of optimizer goals
- Choice of access paths
- Choice of join orders

The optimizer generates most of the possible ways of processing a query and assigns a cost to each step in the generated execution plan. The plan with the lowest cost is chosen as the query plan to be executed.

> **Note:**
>
> You can obtain an execution plan for a SQL statement without executing the plan. Only an execution plan that the database actually uses to execute a query is correctly termed a query plan.

You can influence optimizer choices by setting the optimizer goal and by gathering representative statistics for the optimizer. For example, you may set the optimizer goal to either of the following:

- Total throughput

  The `ALL_ROWS` hint instructs the optimizer to get the last row of the result to the client application as fast as possible.

- Initial response time

  The `FIRST_ROWS` hint instructs the optimizer to get the first row to the client as fast as possible.

A typical end-user, interactive application would benefit from initial response time optimization, whereas a batch-mode, non-interactive application would benefit from total throughput optimization.

**See Also:**

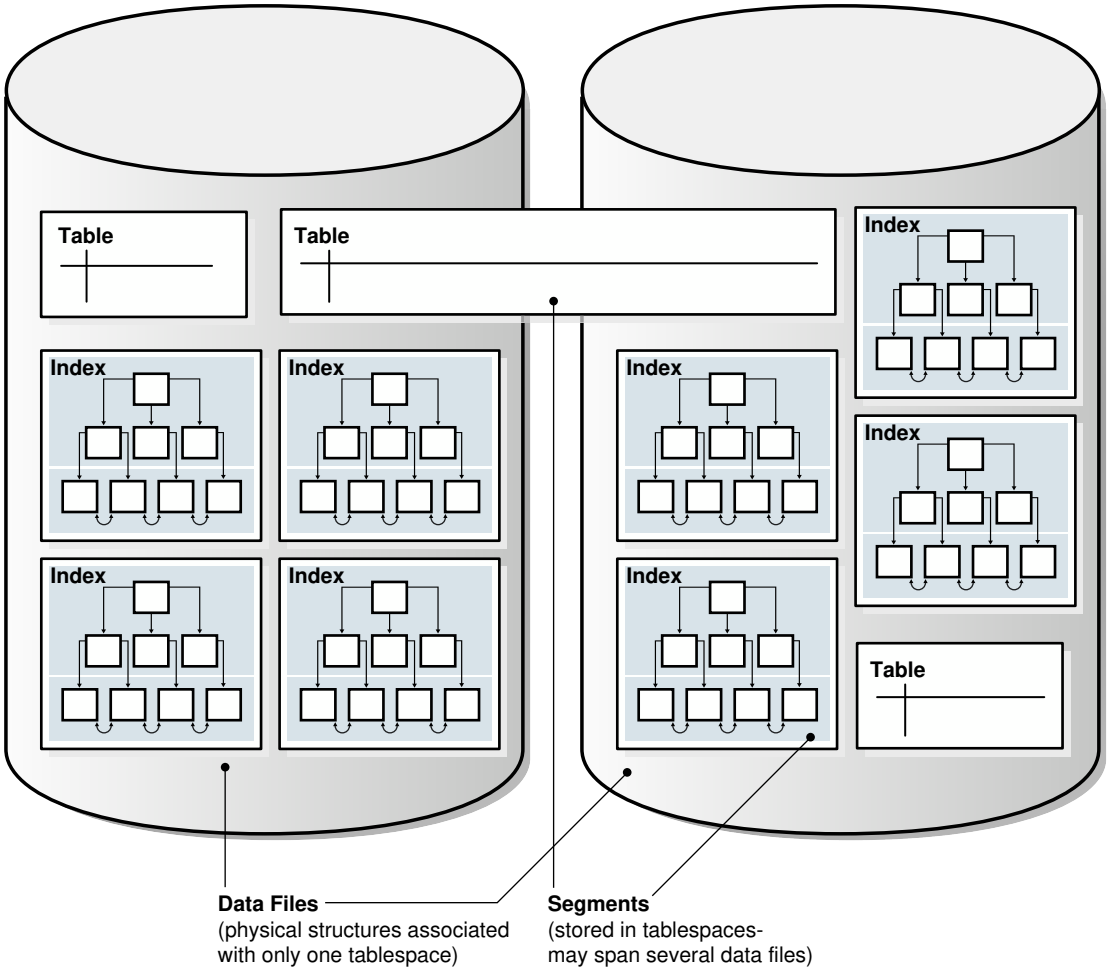
- *Oracle Database PL/SQL Packages and Types Reference* for information about using `DBMS_STATS`
- *Oracle Database SQL Tuning Guide* for more information about the optimizer and using hints

## Optimizer Components

The optimizer contains three main components: the transformer, estimator, and plan generator.

The following diagram depicts the components:

**Figure 9-2 Optimizer Components**




The input to the optimizer is a parsed query. The optimizer performs the following operations:

1. The optimizer receives the parsed query and generates a set of potential plans for the SQL statement based on available access paths and hints.
2. The optimizer estimates the cost of each plan based on statistics in the data dictionary. The cost is an estimated value proportional to the expected resource use needed to execute the statement with a particular plan.
3. The optimizer compares the costs of plans and chooses the lowest-cost plan, known as the query plan, to pass to the row source generator.

- Query Transformer
  The **query transformer** determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan. The input to the query transformer is a parsed query, which the optimizer represents as a set of query blocks.
- Estimator
  The **estimator** determines the overall cost of a given execution plan.
- Plan Generator
  The **plan generator** tries out different plans for a submitted query. The optimizer chooses the plan with the lowest cost.

> **See Also:**
>
> - "SQL Parsing"
> - "SQL Row Source Generation"

## Query Transformer

The **query transformer** determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan. The input to the query transformer is a parsed query, which the optimizer represents as a set of query blocks.

> **See Also:**
>
> "Query Rewrite"

## Estimator

The **estimator** determines the overall cost of a given execution plan.

The estimator generates three different types of measures to achieve this goal:

- Selectivity

  This measure represents a fraction of rows from a row set. The selectivity is tied to a query predicate, such as `last_name='Smith'`, or a combination of predicates.

- Cardinality

  This measure represents the number of rows in a row set.

- Cost

  This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

If statistics are available, then the estimator uses them to compute the measures. The statistics improve the degree of accuracy of the measures.

## Plan Generator

The **plan generator** tries out different plans for a submitted query. The optimizer chooses the plan with the lowest cost.

For each nested subquery and unmerged view, the optimizer generates a subplan. The optimizer represents each subplan as a separate query block. The plan generator explores various plans for a query block by trying out different access paths, join methods, and join orders.

The adaptive query optimization capability changes plans based on statistics collected during statement execution. All adaptive mechanisms can execute a final plan for a statement that differs from the default plan. Adaptive optimization uses either dynamic plans, which choose among subplans during statement execution, or reoptimization, which changes a plan on executions after the current execution.

> **See Also:**
>
> - *Oracle Database 2 Day + Performance Tuning Guide* for an introduction to SQL tuning
> - *Oracle Database SQL Tuning Guide* to learn about the optimizer components and adaptive optimization

## Access Paths

An **access path** is the technique that a query uses to retrieve rows.

For example, a query that uses an index has a different access path from a query that does not. In general, index access paths are best for statements that retrieve a small subset of table rows. Full scans are more efficient for accessing a large portion of a table.

The database can use several different access paths to retrieve data from a table. The following is a representative list:

- Full table scans

  This type of scan reads all rows from a table and filters out those that do not meet the selection criteria. The database sequentially scans all data blocks in the segment, including those under the high water mark (HWM) that separates used from unused space (see "Segment Space and the High Water Mark").

- Rowid scans

  The rowid of a row specifies the data file and data block containing the row and the location of the row in that block. The database first obtains the rowids of the selected rows, either from the statement `WHERE` clause or through an index scan, and then locates each selected row based on its rowid.

- Index scans

  This scan searches an index for the indexed column values accessed by the SQL statement (see "Index Scans"). If the statement accesses only columns of the index, then Oracle Database reads the indexed column values directly from the index.

- Cluster scans

  A cluster scan retrieves data from a table stored in an indexed table cluster, where all rows with the same cluster key value are stored in the same data block (see "Overview of Indexed Clusters"). The database first obtains the rowid of a selected row by scanning the cluster index. Oracle Database locates the rows based on this rowid.

- Hash scans

  A hash scan locates rows in a hash cluster, where all rows with the same hash value are stored in the same data block (see "Overview of Hash Clusters"). The database first obtains the hash value by applying a hash function to a cluster key value specified by the statement. Oracle Database then scans the data blocks containing rows with this hash value.

The optimizer chooses an access path based on the available access paths for the statement and the estimated cost of using each access path or combination of paths.

> **See Also:**
>
> *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database SQL Tuning Guide* to learn about access paths

## Optimizer Statistics

The **optimizer statistics** are a collection of data that describe details about the database and the objects in the database. The statistics provide a statistically correct picture of data storage and distribution usable by the optimizer when evaluating access paths.

Optimizer statistics include the following:

- Table statistics

  These include the number of rows, number of blocks, and average row length.
- Column statistics

  These include the number of distinct values and nulls in a column and the distribution of data.
- Index statistics

  These include the number of leaf blocks and index levels.
- System statistics

  These include CPU and I/O performance and utilization.

Oracle Database gathers optimizer statistics on all database objects automatically and maintains these statistics as an automated maintenance task. You can also gather statistics manually using the `DBMS_STATS` package. This PL/SQL package can modify, view, export, import, and delete statistics.

> **Note:**
>
> Optimizer statistics are created for the purposes of query optimization and are stored in the data dictionary. Do not confuse these statistics with performance statistics visible through dynamic performance views.

Optimizer Statistics Advisor is built-in diagnostic software that analyzes how you are currently gathering statistics, the effectiveness of existing statistics gathering jobs, and the quality of the gathered statistics. Optimizer Statistics Advisor maintains rules, which embody Oracle best practices based on the current feature set. In this way, the advisor always provides the most up-to-date recommendations for statistics gathering.

> **See Also:**
>
> - *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database SQL Tuning Guide* to learn how to gather and manage statistics
> - *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_STATS`

# Optimizer Hints

A **hint** is a comment in a SQL statement that acts as an instruction to the optimizer.

Sometimes the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to run a SQL statement. The application designer can use hints in SQL statements to specify how the statement should be run. The following examples illustrate the use of hints.

**Example 9-2 Execution Plan for SELECT with FIRST_ROWS Hint**

Suppose that your interactive application runs a query that returns 50 rows. This application initially fetches only the first 25 rows of the query to present to the end user. You want the optimizer to generate a plan that gets the first 25 records as quickly as possible so that the user is not forced to wait. You can use a hint to pass this instruction to the optimizer as shown in the `SELECT` statement and `AUTOTRACE` output in the following example:

```
SELECT /*+ FIRST_ROWS(25) */ employee_id, department_id
FROM   hr.employees
WHERE  department_id > 50;

------------------------------------------------------------------------------
| Id | Operation                    | Name              | Rows | Bytes
------------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |                   | 26   | 182
|  1 |  TABLE ACCESS BY INDEX ROWID | EMPLOYEES         | 26   | 182
|* 2 |   INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |      |
------------------------------------------------------------------------------
```

In this example, the execution plan shows that the optimizer chooses an index on the `employees.department_id` column to find the first 25 rows of `employees` whose department ID is over 50. The optimizer uses the rowid retrieved from the index to retrieve the record from the `employees` table and return it to the client. Retrieval of the first record is typically almost instantaneous.

**Example 9-3 Execution Plan for SELECT with No Hint**

Assume that you execute the same statement, but without the optimizer hint:

```
SELECT employee_id, department_id
FROM   hr.employees
WHERE  department_id > 50;

------------------------------------------------------------------------------
| Id | Operation             | Name              | Rows | Bytes | Cos
------------------------------------------------------------------------------
|  0 | SELECT STATEMENT      |                   | 50   | 350   |
|* 1 |  VIEW                 | index$_join$_001  | 50   | 350   |
|* 2 |   HASH JOIN           |                   |      |       |
```

```
|* 3 |    INDEX RANGE SCAN     | EMP_DEPARTMENT_IX | 50   | 350   |
|  4 |    INDEX FAST FULL SCAN| EMP_EMP_ID_PK     | 50   | 350   |
```

In this case, the execution plan joins two indexes to return the requested records as fast as possible. Rather than repeatedly going from index to table as in Example 9-2, the optimizer chooses a range scan of `EMP_DEPARTMENT_IX` to find all rows where the department ID is over 50 and place these rows in a hash table. The optimizer then chooses to read the `EMP_EMP_ID_PK` index. For each row in this index, it probes the hash table to find the department ID.

In this case, the database cannot return the first row to the client until the index range scan of `EMP_DEPARTMENT_IX` completes. Thus, this generated plan would take longer to return the first record. Unlike the plan in Example 9-2, which accesses the table by index rowid, the plan uses multiblock I/O, resulting in large reads. The reads enable the last row of the entire result set to be returned more rapidly.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* to learn how to use optimizer hints

# Overview of SQL Processing

This section explains how Oracle Database processes SQL statements. Specifically, the section explains the way in which the database processes DDL statements to create objects, DML to modify data, and queries to retrieve data.

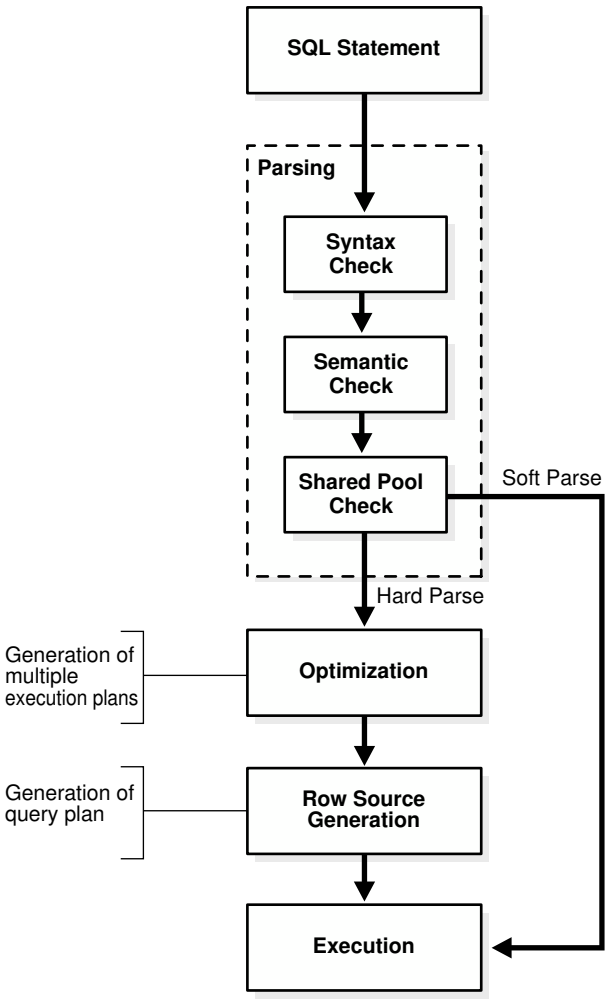
- Stages of SQL Processing
  The general stages of SQL processing are parsing, optimization, row source generation, and execution. Depending on the statement, the database may omit some of these steps.
- Differences Between DML and DDL Processing
  Oracle Database processes DDL differently from DML.

## Stages of SQL Processing

The general stages of SQL processing are parsing, optimization, row source generation, and execution. Depending on the statement, the database may omit some of these steps.

The following figure depicts the general stages:

**Figure 9-3 Stages of SQL Processing**




- SQL Parsing
  The first stage of SQL processing is **SQL parsing**. This stage involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.
- SQL Optimization
  **Query optimization** is the process of choosing the most efficient means of executing a SQL statement.
- SQL Row Source Generation
  The **row source generator** is software that receives the optimal execution plan from the optimizer and produces an iterative plan, called the **query plan**, that is usable by the rest of the database.
- SQL Execution
  During execution, the SQL engine executes each row source in the tree produced by the row source generator. This is the only mandatory step in DML processing.

## SQL Parsing

The first stage of SQL processing is **SQL parsing**. This stage involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.

When an application issues a SQL statement, the application makes a parse call to the database to prepare the statement for execution. The parse call opens or creates a cursor, which is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information. The cursor and private SQL area are in the PGA.

During the parse call, the database performs the following checks:

- Syntax check
- Semantic check
- Shared pool check

The preceding checks identify the errors that can be found *before statement execution*. Some errors cannot be caught by parsing. For example, the database can encounter a deadlock or errors in data conversion only during statement execution.

> **See Also:**
>
> "Locks and Deadlocks"

## SQL Optimization

**Query optimization** is the process of choosing the most efficient means of executing a SQL statement.

The database optimizes queries based on statistics collected about the actual data being accessed. The optimizer uses the number of rows, the size of the data set, and other factors to generate possible execution plans, assigning a numeric cost to each plan. The database uses the plan with the lowest cost.

The database must perform a hard parse at least once for every unique DML statement and performs optimization during this parse. DDL is never optimized unless it includes a DML component such as a subquery that requires optimization.

> **See Also:**
>
> - "Overview of the Optimizer"
> - *Oracle Database SQL Tuning Guide* for detailed information about the query optimizer

## SQL Row Source Generation

The **row source generator** is software that receives the optimal execution plan from the optimizer and produces an iterative plan, called the **query plan**, that is usable by the rest of the database.

The query plan takes the form of a combination of steps. Each step returns a row set. The rows in this set are either used by the next step or, in the last step, are returned to the application issuing the SQL statement.

A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows. The row source can be a table, view, or result of a join or grouping operation.

## SQL Execution

During execution, the SQL engine executes each row source in the tree produced by the row source generator. This is the only mandatory step in DML processing.

During execution, if the data is not in memory, then the database reads the data from disk into memory. The database also takes out any locks and latches necessary to ensure data integrity and logs any changes made during the SQL execution. The final stage of processing a SQL statement is closing the cursor.

If the database is configured to use In-Memory Column Store (IM column store), then the database transparently routes queries to the IM column store when possible, and to disk and the database buffer cache otherwise. A single query can also use the IM column store, disk, and the buffer cache. For example, a query might join two tables, only one of which is cached in the IM column store.

**See Also:**

- "In-Memory Area"
- *Oracle Database SQL Tuning Guide* for detailed information about execution plans and the `EXPLAIN PLAN` statement

# Differences Between DML and DDL Processing

Oracle Database processes DDL differently from DML.

For example, when you create a table, the database does not optimize the `CREATE TABLE` statement. Instead, Oracle Database parses the DDL statement and carries out the command.

In contrast to DDL, most DML statements have a query component. In a query, execution of a cursor places the row generated by the query into the result set.

The database can fetch result set rows either one row at a time or in groups. In the fetch, the database selects rows and, if requested by the query, sorts the rows. Each successive fetch retrieves another row of the result until the last row has been fetched.

**See Also:**

*Oracle Database Development Guide* to learn about processing DDL, transaction control, and other types of statements

10

# Server-Side Programming: PL/SQL and Java

**SQL** explains the Structured Query Language (SQL) language and how the database processes SQL statements. This chapter explains how Procedural Language/SQL (PL/SQL) or Java programs stored in the database can use SQL.

This chapter includes the following topics:


- Introduction to Server-Side Programming
- Overview of PL/SQL
- Overview of Java in Oracle Database
- Overview of Triggers


- Introduction to Server-Side Programming
  In a nonprocedural language such as SQL, the set of data to be operated on is specified, but not the operations to be performed or the manner in which they are to be carried out.
- Overview of PL/SQL
  PL/SQL provides a server-side, stored procedural language that is easy-to-use, seamless with SQL, robust, portable, and secure. You can access and manipulate database data using procedural objects called **PL/SQL units**.
- Overview of Java in Oracle Database
  Java has emerged as the object-oriented programming language of choice.
- Overview of Triggers
  A database **trigger** is a compiled stored program unit, written in either PL/SQL or Java, that Oracle Database invokes ("fires") automatically in certain situations.

> **See Also:**
>
> "SQL" for an overview of the SQL language

## Introduction to Server-Side Programming

In a nonprocedural language such as SQL, the set of data to be operated on is specified, but not the operations to be performed or the manner in which they are to be carried out.

In a procedural language program, most statement execution depends on previous or subsequent statements and on control structures, such as loops or conditional branches, that are not available in SQL. For an illustration of the difference between procedural and nonprocedural languages, suppose that the following SQL statement queries the `employees` table:

```
SELECT employee_id, department_id, last_name, salary FROM employees;
```

The preceding statement requests data, but does not apply logic to the data. However, suppose you want an application to determine whether each employee in the data set deserves a raise based on salary and department performance. A necessary condition of a raise is that the employee did not receive more than three raises in the last five years. If a raise is called for, then the application must adjust the salary and email the manager; otherwise, the application must update a report.

The problem is how procedural database applications requiring conditional logic and program flow control can use SQL. The basic development approaches are as follows:

- Use client-side programming to embed SQL statements in applications written in procedural languages such as C, C++, or Java

  You can place SQL statements in source code and submit it to a precompiler or Java translator before compilation. Alternatively, you can eliminate the precompilation step and use an API such as Java Database Connectivity (JDBC) or Oracle Call Interface (OCI) to enable the application to interact with the database.

- Use server-side programming to develop data logic that resides in the database

  An application can explicitly invoke stored subprograms (procedures and functions), written in PL/SQL (pronounced *P L sequel*) or Java. You can also create a trigger, which is named program unit that is stored in the database and invoked in response to a specified event.

This chapter explains the second approach. The principal benefit of server-side programming is that functionality built into the database can be deployed anywhere. The database and not the application determines the best way to perform tasks on a given operating system. Also, subprograms increase scalability by centralizing application processing on the server, enabling clients to reuse code. Because subprogram calls are quick and efficient, a single call can start a compute-intensive stored subprogram, reducing network traffic.

You can use the following languages to store data logic in Oracle Database:

- PL/SQL

  PL/SQL is the Oracle Database procedural extension to SQL. PL/SQL is integrated with the database, supporting all Oracle SQL statements, functions, and data types. Applications written in database APIs can invoke PL/SQL stored subprograms and send PL/SQL code blocks to the database for execution.

- Java

  Oracle Database also provides support for developing, storing, and deploying Java applications. Java stored subprograms run in the database and are independent of programs that run in the middle tier. Java stored subprograms interface with SQL using a similar execution model to PL/SQL.

**See Also:**

- *Oracle Database Development Guide* to learn about embedding SQL with precompilers and APIs
- *Oracle Database 2 Day Developer's Guide* for an introduction to Oracle Database application development
- *Oracle Database Development Guide* to learn how to choose a programming environment

# Overview of PL/SQL

PL/SQL provides a server-side, stored procedural language that is easy-to-use, seamless with SQL, robust, portable, and secure. You can access and manipulate database data using procedural objects called **PL/SQL units**.

PL/SQL units generally are categorized as follows:

- A PL/SQL subprogram is a PL/SQL block that is stored in the database and can be called by name from an application. When you create a subprogram, the database parses the subprogram and stores its parsed representation in the database. You can declare a subprogram as a procedure or a function.
- A PL/SQL anonymous block is a PL/SQL block that appears in your application and is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear.

The PL/SQL compiler and interpreter are embedded in Oracle SQL Developer, giving developers a consistent and leveraged development model on both client and server. Also, PL/SQL stored procedures can be called from several database clients, such as Pro*C, JDBC, ODBC, or OCI, and from Oracle Reports and Oracle Forms.

- PL/SQL Subprograms
  A PL/SQL subprogram is a named PL/SQL block that permits the caller to supply parameters that can be input only, output only, or input and output values.
- PL/SQL Packages
  A **PL/SQL package** is a group of related subprograms, along with the cursors and variables they use, stored together in the database for continued use as a unit. Packaged subprograms can be called explicitly by applications or users.
- PL/SQL Anonymous Blocks
  A PL/SQL anonymous block is an unnamed, nonpersistent PL/SQL unit.
- PL/SQL Language Constructs
  PL/SQL blocks can include a variety of different PL/SQL language constructs.
- PL/SQL Collections and Records
  Many programming techniques use collection types such as arrays, bags, lists, nested tables, sets, and trees.
- How PL/SQL Runs
  PL/SQL supports both interpreted execution and native execution.

**See Also:**

- *Oracle Database 2 Day Developer's Guide*
- *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL, including packages

## PL/SQL Subprograms

A PL/SQL subprogram is a named PL/SQL block that permits the caller to supply parameters that can be input only, output only, or input and output values.

A subprogram solves a specific problem or performs related tasks and serves as a building block for modular, maintainable database applications. A subprogram is either a PL/SQL procedure or a PL/SQL function. Procedures and functions are identical except that functions always return a single value to the caller, whereas procedures do not. The term *PL/SQL procedure* in this chapter refers to either a procedure or a function.

- Advantages of PL/SQL Subprograms
  Server-side programming has many advantages over client-side programming.
- Creation of PL/SQL Subprograms
  A standalone stored subprogram is a subprogram created at the schema level with the `CREATE PROCEDURE` or `CREATE FUNCTION` statement. Subprograms defined in a package are called **package subprograms** and are considered a part of the package.
- Execution of PL/SQL Subprograms
  Users can execute a subprogram interactively in multiple ways.

> **See Also:**
>
> - *Pro*C/C++ Programmer's Guide* and *Pro*COBOL Programmer's Guide* to learn about stored procedures in these languages
> - *Oracle Database PL/SQL Language Reference*

## Advantages of PL/SQL Subprograms

Server-side programming has many advantages over client-side programming.

Advantages include:

- Improved performance
  - The amount of information that an application must send over a network is small compared with issuing individual SQL statements or sending the text of an entire PL/SQL block to Oracle Database, because the information is sent only once and thereafter invoked when it is used.
  - The compiled form of a procedure is readily available in the database, so no compilation is required at execution time.
  - If the procedure is present in the shared pool of the SGA, then the database need not retrieve it from disk and can begin execution immediately.
- Memory allocation

  Because stored procedures take advantage of the shared memory capabilities of Oracle Database, it must load only a single copy of the procedure into memory for execution by multiple users. Sharing code among users results in a substantial reduction in database memory requirements for applications.
- Improved productivity

  Stored procedures increase development productivity. By designing applications around a common set of procedures, you can avoid redundant coding. For example, you can write procedures to manipulate rows in the `employees` table. Any application can call these procedures without requiring SQL statements to be rewritten. If the methods of data management change, then only the procedures must be modified, not the applications that use the procedures.

Stored procedures are perhaps the best way to achieve code reuse. Because any client application written in any language that connects to the database can invoke stored procedures, they provide maximum code reuse in all environments.

- Integrity

  Stored procedures improve the integrity and consistency of your applications. By developing applications around a common group of procedures, you reduce the likelihood of coding errors.

  For example, you can test a subprogram to guarantee that it returns an accurate result and, after it is verified, reuse it in any number of applications without retesting. If the data structures referenced by the procedure are altered, then you must only recompile the procedure. Applications that call the procedure do not necessarily require modifications.

- Security with definer's rights procedures

  Stored procedures can help enforce data security. A definer's rights PL/SQL procedure executes with the privilege of its owner, not its current user. Thus, you can restrict the database tasks that users perform by allowing them to access data only through procedures and functions that run with the definer's privileges.

  For example, you can grant users access to a procedure that updates a table but not grant access to the table itself. When a user invokes the procedure, it runs with the privileges of its owner. Users who have only the privilege to run the procedure (but not privileges to query, update, or delete from the underlying tables) can invoke the procedure but not manipulate table data in any other way.

- Inherited privileges and schema context with invoker's rights procedures

  An invoker's rights PL/SQL procedure executes in the current user's schema with the current user's privileges. In other words, an invoker's rights procedure is not tied to a particular user or schema. Invoker's rights procedures make it easy for application developers to centralize application logic, even when the underlying data is divided among user schemas.

  For example, an `hr_manager` user who runs an update procedure on the `hr.employees` table can update salaries, whereas an `hr_clerk` who runs the same procedure is restricted to updating address data.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for an overview of PL/SQL subprograms
- *Oracle Database Security Guide* to learn more about definer's and invoker's rights

## Creation of PL/SQL Subprograms

A standalone stored subprogram is a subprogram created at the schema level with the `CREATE PROCEDURE` or `CREATE FUNCTION` statement. Subprograms defined in a package are called **package subprograms** and are considered a part of the package.

The database stores subprograms in the data dictionary as schema objects. A subprogram has a specification, which includes descriptions of any parameters, and a body.

**Example 10-1 PL/SQL Procedure**

```
hire_employeesemployees

CREATE PROCEDURE hire_employees
  (p_last_name VARCHAR2, p_job_id VARCHAR2, p_manager_id NUMBER,
   p_hire_date DATE, p_salary NUMBER, p_commission_pct NUMBER,
   p_department_id NUMBER)
IS
BEGIN
.
.
.
  INSERT INTO employees (employee_id, last_name, job_id, manager_id,
hire_date,
    salary, commission_pct, department_id)
  VALUES (emp_sequence.NEXTVAL, p_last_name, p_job_id, p_manager_id,
          p_hire_date, p_salary, p_commission_pct, p_department_id);
.
.
.
END;
```

**See Also:**

- *Oracle Database 2 Day Developer's Guide* to learn how to create subprograms
- *Oracle Database PL/SQL Language Reference* to learn about the `CREATE PROCEDURE` statement

## Execution of PL/SQL Subprograms

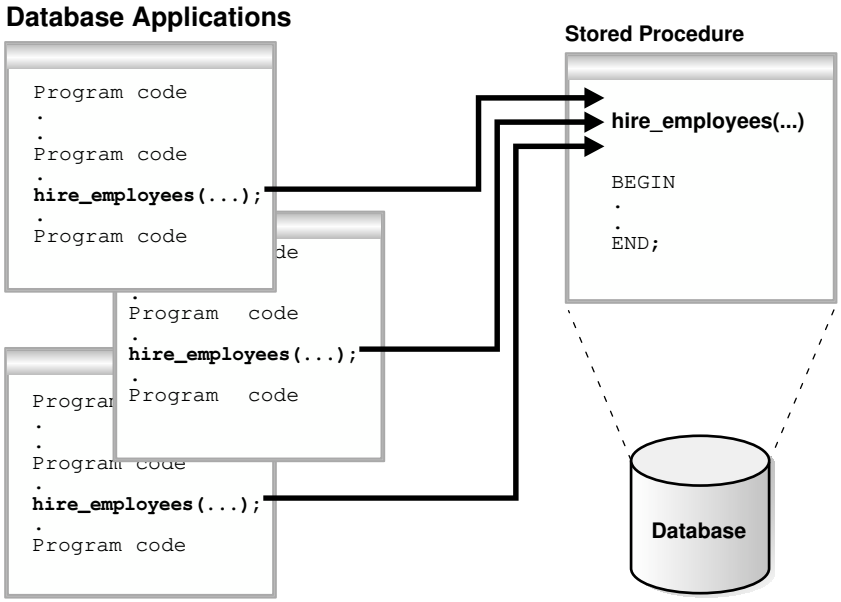
Users can execute a subprogram interactively in multiple ways.

The options are:

- Using an Oracle tool, such as SQL*Plus or SQL Developer
- Calling it explicitly in the code of a database application, such as an Oracle Forms or precompiler application
- Calling it explicitly in the code of another procedure or trigger

The following graphic shows different database applications calling `hire_employees`.

**Figure 10-1 Multiple Executions of a Stored Procedure**




Alternatively, a privileged user can use Oracle Enterprise Manager or SQL*Plus to run the `hire_employees` procedure using a statement such as the following:

```
EXECUTE hire_employees ('TSMITH', 'CLERK', 1037, SYSDATE, 500, NULL, 20);
```

The preceding statement inserts a new record for `TSMITH` in the `employees` table.

A stored procedure depends on the objects referenced in its body. The database automatically tracks and manages these dependencies. For example, if you alter the definition of the `employees` table referenced by the `hire_employees` procedure in a manner that would affect this procedure, then the procedure must be recompiled to validate that it still works as designed. Usually, the database automatically administers such dependency management.

**See Also:**

- *Oracle Database 2 Day Developer's Guide* to learn more about SQL*Plus and SQL Developer
- *Oracle Database Development Guide* to learn more about precompilers
- *Oracle Database PL/SQL Language Reference* to learn how to use PL/SQL subprograms
- *SQL*Plus User's Guide and Reference* to learn about the `EXECUTE` command

# PL/SQL Packages

A **PL/SQL package** is a group of related subprograms, along with the cursors and variables they use, stored together in the database for continued use as a unit. Packaged subprograms can be called explicitly by applications or users.

Oracle Database includes many supplied packages that extend database functionality and provide PL/SQL access to SQL features. For example, the `UTL_HTTP` package enables HTTP callouts from PL/SQL and SQL to access data on the Internet or to call Oracle Web Server Cartridges. You can use the supplied packages when creating applications or as a source of ideas when creating your own stored procedures.

- Advantages of PL/SQL Packages
  PL/SQL packages provide a number of advantages to the application developer.
- Creation of PL/SQL Packages
  You create a package in two parts: the specification and the body. The package specification declares all public constructs of the package, whereas the package body defines all constructs (public and private) of the package.
- Execution of PL/SQL Package Subprograms
  Database triggers, stored subprograms, 3GL application programs, and Oracle tools can reference package contents.

## Advantages of PL/SQL Packages

PL/SQL packages provide a number of advantages to the application developer.

Advantages include:

- Encapsulation

  Packages enable you to encapsulate or group stored procedures, variables, data types, and so on in a named, stored unit. Encapsulation provides better organization during development and also more flexibility. You can create specifications and reference public procedures without actually creating the package body. Encapsulation simplifies privilege management. Granting the privilege for a package makes package constructs accessible to the grantee.

- Data security

  The methods of package definition enable you to specify which variables, cursors, and procedures are public and private. Public means that it is directly accessible to the user of a package. Private means that it is hidden from the user of a package.

  For example, a package can contain 10 procedures. You can define the package so that only three procedures are public and therefore available for execution by a user of the package. The remaining procedures are private and can only be accessed by the procedures within the package. Do not confuse public and private package variables with grants to `PUBLIC`.

- Better performance

  An entire package is loaded into memory in small chunks when a procedure in the package is called for the first time. This load is completed in one operation, as opposed to the separate loads required for standalone procedures. When calls to related packaged procedures occur, no disk I/O is needed to run the compiled code in memory.

  A package body can be replaced and recompiled without affecting the specification. Consequently, schema objects that reference a package's constructs (always through the specification) need not be recompiled unless the package specification is also replaced. By using packages, unnecessary recompilations can be minimized, resulting in less impact on overall database performance.

## Creation of PL/SQL Packages

You create a package in two parts: the specification and the body. The package specification declares all public constructs of the package, whereas the package body defines all constructs (public and private) of the package.

The following example shows part of a statement that creates the package specification for `employees_management`, which encapsulates several subprograms used to manage an employee database. Each part of the package is created with a different statement.

```
CREATE PACKAGE employees_management AS
  FUNCTION hire_employees (last_name VARCHAR2, job_id VARCHAR2, manager_id
NUMBER,
    salary NUMBER, commission_pct NUMBER, department_id NUMBER) RETURN
NUMBER;
  PROCEDURE fire_employees(employee_id NUMBER);
  PROCEDURE salary_raise(employee_id NUMBER, salary_incr NUMBER);
.
.
.
  no_sal EXCEPTION;
END employees_management;
```

The specification declares the function `hire_employees`, the procedures `fire_employees` and `salary_raise`, and the exception `no_sal`. All of these public program objects are available to users who have access to the package.

The `CREATE PACKAGE BODY` statement defines objects declared in the specification. The package body must be created in the same schema as the package. After creating the package, you can develop applications that call any of these public procedures or functions or raise any of the public exceptions of the package.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* to learn about the `CREATE PACKAGE` statement

## Execution of PL/SQL Package Subprograms

Database triggers, stored subprograms, 3GL application programs, and Oracle tools can reference package contents.

The following graphic shows database applications invoking procedures and functions in the `employees_management` package.

**Figure 10-2 Calling Subprograms in a PL/SQL Package**

Database applications explicitly call packaged procedures as necessary. After being granted the privileges for the `employees_management` package, a user can explicitly run any of the procedures contained in it. For example, SQL*Plus can issue the following statement to run the `hire_employees` package procedure:

```
EXECUTE employees_management.hire_employees ('TSMITH', 'CLERK', 1037,
SYSDATE, 500, NULL, 20);
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for an introduction to PL/SQL packages
- *Oracle Database Development Guide* to learn how to code PL/SQL packages

# PL/SQL Anonymous Blocks

A PL/SQL anonymous block is an unnamed, nonpersistent PL/SQL unit.

Typical uses for anonymous blocks include:

- Initiating calls to subprograms and package constructs
- Isolating exception handling

- Managing control by nesting code within other PL/SQL blocks

Anonymous blocks do not have the code reuse advantages of stored subprograms. Table 10-1 summarizes the differences between the two types of PL/SQL units.

**Table 10-1 Differences Between Anonymous Blocks and Subprograms**

| Is the PL/SQL Unit ... | Anonymous Blocks | Subprograms |
|---|---|---|
| Specified with a name? | No | Yes |
| Compiled with every reuse? | No | No |
| Stored in the database? | No | Yes |
| Invocable by other applications? | No | Yes |
| Capable of returning bind variable values? | Yes | Yes |
| Capable of returning function values? | No | Yes |
| Capable of accepting parameters? | No | Yes |

An anonymous block consists of an optional declarative part, an executable part, and one or more optional exception handlers. The following sample anonymous block selects an employee last name into a variable and prints the name:

```
DECLARE
  v_lname VARCHAR2(25);
BEGIN
  SELECT last_name
    INTO v_lname
  FROM employees
  WHERE employee_id = 101;
  DBMS_OUTPUT.PUT_LINE('Employee last name is '||v_lname);
END;
```

Oracle Database compiles the PL/SQL block and places it in the shared pool of the SGA, but it does not store the source code or compiled version in the database for reuse beyond the current instance. Unlike triggers, an anonymous block is compiled each time it is loaded into memory. Shared SQL allows anonymous PL/SQL blocks in the shared pool to be reused and shared until they are flushed out of the shared pool.

> **See Also:**
>
> *Oracle Database Development Guide* to learn more about anonymous PL/SQL blocks

# PL/SQL Language Constructs

PL/SQL blocks can include a variety of different PL/SQL language constructs.

These constructs including the following:

- Variables and constants

You can declare these constructs within a procedure, function, or package. You can use a variable or constant in a SQL or PL/SQL statement to capture or provide a value when one is needed.

- Cursors

  You can declare a cursor explicitly within a procedure, function, or package to facilitate record-oriented processing of Oracle Database data. The PL/SQL engine can also declare cursors implicitly.

- Exceptions

  PL/SQL lets you explicitly handle internal and user-defined error conditions, called *exceptions*, that arise during processing of PL/SQL code.

PL/SQL can run dynamic SQL statements whose complete text is not known until run time. Dynamic SQL statements are stored in character strings that are entered into, or built by, the program at run time. This technique enables you to create general purpose procedures. For example, you can create a procedure that operates on a table whose name is not known until run time.

**See Also:**

- *Oracle Database PL/SQL Packages and Types Reference* for details about dynamic SQL
- *Oracle Database PL/SQL Packages and Types Reference* to learn how to use dynamic SQL in the `DBMS_SQL` package

# PL/SQL Collections and Records

Many programming techniques use collection types such as arrays, bags, lists, nested tables, sets, and trees.

To support collection techniques in database applications, PL/SQL provides the data types `TABLE` and `VARRAY`. These types enable you to declare associative arrays, nested tables, and variable-size arrays.

- Collections
  A **PL/SQL collection** is an ordered group of elements, all of the same type.
- Records
  A **PL/SQL record** is a composite variable that can store data values of different types, similar to a struct type in C, C++, or Java. Records are useful for holding data from table rows, or certain columns from table rows.

## Collections

A **PL/SQL collection** is an ordered group of elements, all of the same type.

Each element has a unique subscript that determines its position in the collection. To create a collection, you first define a collection type, and then declare a variable of that type.

Collections work like the arrays found in most third-generation programming languages. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

## Records

A **PL/SQL record** is a composite variable that can store data values of different types, similar to a struct type in C, C++, or Java. Records are useful for holding data from table rows, or certain columns from table rows.

Suppose you have data about an employee such as name, salary, and hire date. These items are dissimilar in type but logically related. A record containing a field for each item lets you treat the data as a logical unit.

You can use the `%ROWTYPE` attribute to declare a record that represents a table row or row fetched from a cursor. With user-defined records, you can declare your own fields.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* to learn how to use PL/SQL records

# How PL/SQL Runs

PL/SQL supports both interpreted execution and native execution.

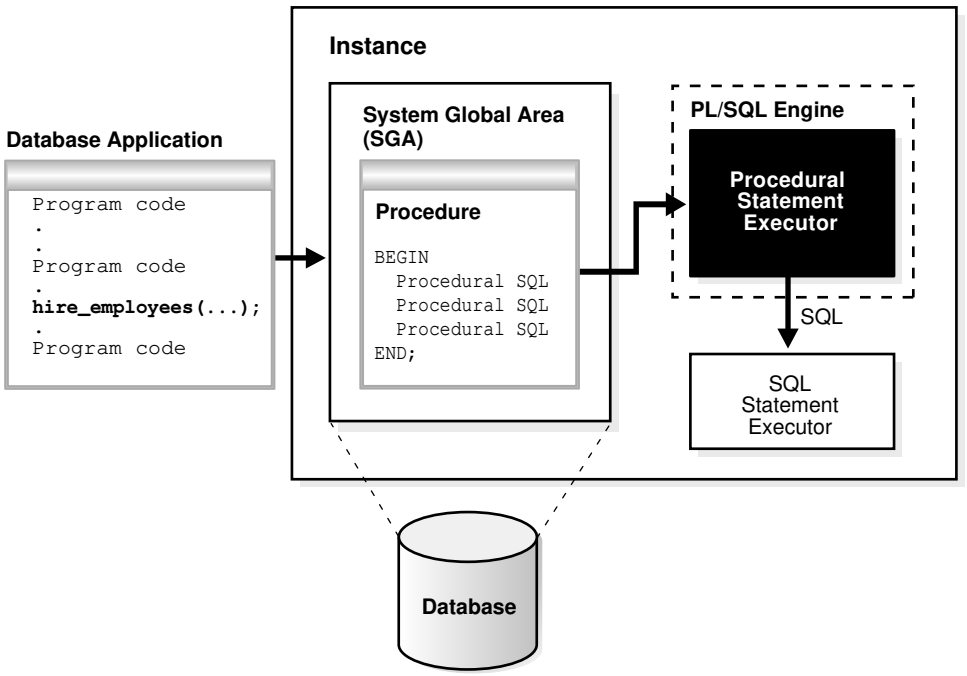
In interpreted execution, PL/SQL source code is compiled into a so-called bytecode representation. A portable virtual computer implemented as part of Oracle Database runs this bytecode.

Native execution offers the best performance on computationally intensive units. In this case, the source code of PL/SQL units is compiled directly to object code for the given platform. This object code is linked into Oracle Database.

The PL/SQL engine defines, compiles, and runs PL/SQL units. This engine is a special component of many Oracle products, including Oracle Database. While many Oracle products have PL/SQL components, this topic specifically covers the PL/SQL units that can be stored in Oracle Database and processed using Oracle Database PL/SQL engine. The documentation for each Oracle tool describes its PL/SQL capabilities.

The following graphic illustrates the PL/SQL engine contained in Oracle Database.

**Figure 10-3 The PL/SQL Engine and Oracle Database**




The PL/SQL unit is stored in a database. When an application calls a stored procedure, the database loads the compiled PL/SQL unit into the shared pool in the system global area (SGA). The PL/SQL and SQL statement executors work together to process the statements in the procedure.

You can call a stored procedure from another PL/SQL block, which can be either an anonymous block or another stored procedure. For example, you can call a stored procedure from Oracle Forms.

A PL/SQL procedure executing on Oracle Database can call an external procedure or function written in the C programming language and stored in a shared library. The C routine runs in a separate address space from that of Oracle Database.

**See Also:**

- "Shared Pool " to learn more about the purpose and contents of the shared pool
- *Oracle Database PL/SQL Language Reference* to learn about PL/SQL architecture
- *Oracle Database Development Guide* to learn more about external procedures

# Overview of Java in Oracle Database

Java has emerged as the object-oriented programming language of choice.

Java includes the following features:

- A Java Virtual Machine (JVM), which provides the basis for platform independence

- Automated storage management techniques, such as garbage collection
- Language syntax that borrows from C and enforces strong typing

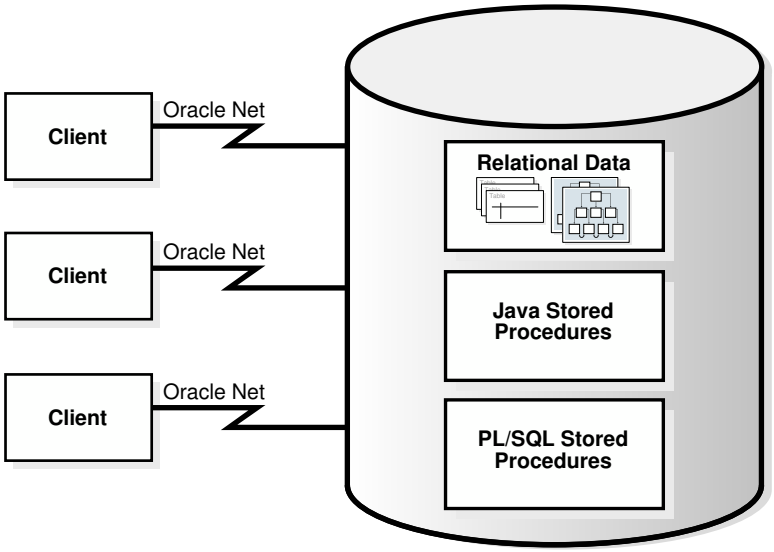
> **Note:**
>
> This chapter assumes that you have some familiarity with the Java language.

The database provides Java programs with a dynamic data-processing engine that supports complex queries and multiple views of data. Client requests are assembled as data queries for immediate processing. Query results are generated dynamically.

The combination of Java and Oracle Database helps you create component-based, network-centric applications that can be easily updated as business needs change. In addition, you can move applications and data stores off the desktop and onto intelligent networks and network-centric servers. More importantly, you can access these applications and data stores from any client device.

The following figure shows a traditional two-tier, client/server configuration in which clients call Java stored procedures in the same way that they call PL/SQL subprograms.

**Figure 10-4 Two-Tier Client/Server Configuration**




- Overview of the Java Virtual Machine (JVM)
  A **JVM** is a virtual processor that runs compiled Java code.
- Java Programming Environment
  Oracle furnishes enterprise application developers with an end-to-end Java solution for creating, deploying, and managing Java applications.

> **See Also:**
>
> *Oracle Database 2 Day + Java Developer's Guide* for an introduction to using Java with Oracle Database

# Overview of the Java Virtual Machine (JVM)

A **JVM** is a virtual processor that runs compiled Java code.

Java source code compiles to low-level machine instructions, known as *bytecodes*, that are platform independent. The Java bytecodes are interpreted through the JVM into platform-dependent actions.

- Overview of Oracle JVM
  The **Oracle JVM** is a standard, Java-compatible environment that runs any pure Java application. It is compatible with the JLS and the JVM specifications.
- Main Components of Oracle JVM
  Oracle JVM runs in the same process space and address space as the database kernel by sharing its memory heaps and directly accessing its relational data. This design optimizes memory use and increases throughput.

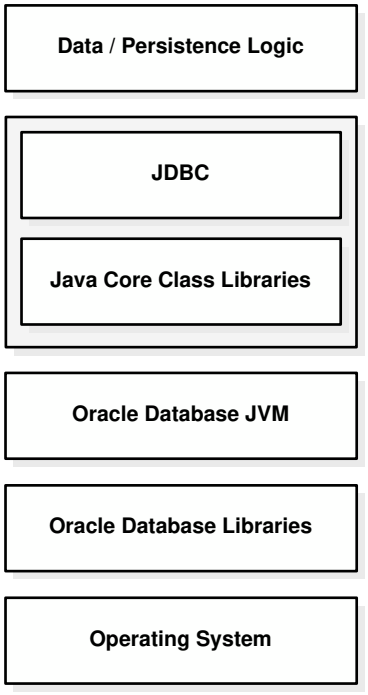
## Overview of Oracle JVM

The **Oracle JVM** is a standard, Java-compatible environment that runs any pure Java application. It is compatible with the JLS and the JVM specifications.

The Oracle JVM supports the standard Java binary format and APIs. In addition, Oracle Database adheres to standard Java language semantics, including dynamic class loading at run time.

The following figure illustrates how Oracle Java applications reside on top of the Java core class libraries, which reside on top of the Oracle JVM. Because the Oracle Java support system is located within the database, the JVM interacts with database libraries, instead of directly interacting with the operating system.

**Figure 10-5 Java Component Structure**




Unlike other Java environments, Oracle JVM is embedded within Oracle Database. Some important differences exist between Oracle JVM and typical client JVMs. For example, in a standard Java environment, you run a Java application through the interpreter by issuing the following command on the command line, where *classname* is the name of the class that the JVM interprets first:

```
java classname
```

The preceding command causes the application to run within a process on your operating system. However, if you are not using the command-line interface, then you must load the application into the database, publish the interface, and then run the application within a database data dictionary.

> **See Also:**
>
> See *Oracle Database Java Developer's Guide* for a description of other differences between the Oracle JVM and typical client JVMs

## Main Components of Oracle JVM

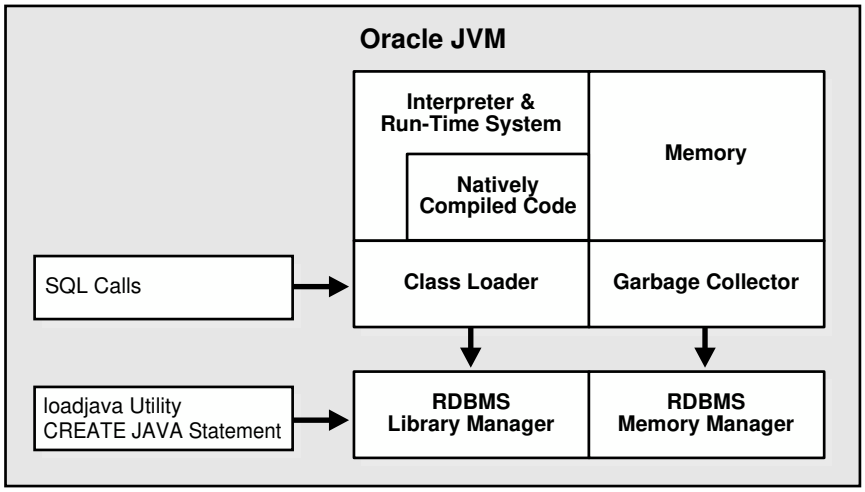
Oracle JVM runs in the same process space and address space as the database kernel by sharing its memory heaps and directly accessing its relational data. This design optimizes memory use and increases throughput.

Oracle JVM provides a run-time environment for Java objects. It fully supports Java data structures, method dispatch, exception handling, and language-level threads. It also supports

all the core Java class libraries, including `java.lang`, `java.io`, `java.net`, `java.math`, and `java.util`.

The following illustration shows the main components of Oracle JVM.

**Figure 10-6 Main Components of Oracle JVM**




Oracle JVM embeds the standard Java namespace in the database schemas. This feature lets Java programs access Java objects stored in Oracle Database and application servers across the enterprise.

In addition, Oracle JVM is tightly integrated with the scalable, shared memory architecture of the database. Java programs use call, session, and object lifetimes efficiently without user intervention. Consequently, Oracle JVM and middle-tier Java business objects can be scaled, even when they have session-long state.

> **See Also:**
>
> *Oracle Database Java Developer's Guide* for a description of the main components of Oracle JVM

## Java Programming Environment

Oracle furnishes enterprise application developers with an end-to-end Java solution for creating, deploying, and managing Java applications.

The solution consists of client-side and server-side programmatic interfaces, tools to support Java development, and a Java Virtual Machine integrated with Oracle Database. All these products are compatible with Java standards.

The Java programming environment consists of the following additional features:

- Java stored procedures as the Java equivalent and companion for PL/SQL. Java stored procedures are tightly integrated with PL/SQL. You can call Java stored procedures from PL/SQL packages and procedures from Java stored procedures.

- The JDBC and SQLJ programming interfaces for accessing SQL data.
- Tools and scripts that assist in developing, loading, and managing classes.
- Java Stored Procedures
  A **Java stored procedure** is a Java method published to SQL and stored in the database.
- Java and PL/SQL Integration
  You can call existing PL/SQL programs from Java and Java programs from PL/SQL. This solution protects and leverages your PL/SQL and Java code.

## Java Stored Procedures

A **Java stored procedure** is a Java method published to SQL and stored in the database.

Like a PL/SQL subprogram, a Java procedure can be invoked directly with products like SQL*Plus or indirectly with a trigger. You can access it from any Oracle Net client—OCI, precompiler, or JDBC.

To publish Java methods, you write call specifications, which map Java method names, parameter types, and return types to their SQL counterparts. When called by client applications, a Java stored procedure can accept arguments, reference Java classes, and return Java result values.

Applications calling the Java method by referencing the name of the call specification. The run-time system looks up the call specification definition in the Oracle data dictionary and runs the corresponding Java method.

In addition, you can use Java to develop powerful programs independently of PL/SQL. Oracle Database provides a fully compliant implementation of the Java programming language and JVM.

> **See Also:**
>
> *Oracle Database Java Developer's Guide* explains how to write stored procedures in Java, how to access them from PL/SQL, and how to access PL/SQL functionality from Java

## Java and PL/SQL Integration

You can call existing PL/SQL programs from Java and Java programs from PL/SQL. This solution protects and leverages your PL/SQL and Java code.

Oracle Database offers two different approaches for accessing SQL data from Java: JDBC and SQLJ. JDBC is available on both client and server, whereas SQLJ is available only on the client.

- JDBC Drivers
  JDBC is a database access protocol that enables you to connect to a database and run SQL statements and queries to the database.
- SQLJ
  **SQLJ** is an ANSI standard for embedding SQL statements in Java programs. You can use client-side SQLJ programs. In addition, you can combine SQLJ programs with JDBC.

## JDBC Drivers

JDBC is a database access protocol that enables you to connect to a database and run SQL statements and queries to the database.

The core Java class libraries provide only one JDBC API, `java.sql`. However, JDBC is designed to enable vendors to supply drivers that offer the necessary specialization for a particular database. Oracle provides the distinct JDBC drivers shown in the following table.

**Table 10-2 JDBC Drivers**

| Driver | Description |
|---|---|
| JDBC Thin driver | You can use the JDBC Thin driver to write pure Java applications and applets that access Oracle SQL data. The JDBC Thin driver is especially well-suited for Web-based applications and applets, because you can dynamically download it from a Web page, similar to any other Java applet. |
| JDBC OCI driver | The JDBC OCI driver accesses Oracle-specific native code, that is, non-Java code, and libraries on the client or middle tier, providing a performance boost compared to the JDBC Thin driver, at the cost of significantly larger size and client-side installation. |
| JDBC server-side internal driver | Oracle Database uses the server-side internal driver when the Java code runs on the server. It allows Java applications running in Oracle JVM on the server to access locally defined data, that is, data on the same system and in the same process, with JDBC. It provides a performance boost, because of its ability to use the underlying Oracle RDBMS libraries directly, without the overhead of an intervening network connection between the Java code and SQL data. By supporting the same Java-SQL interface on the server, Oracle Database does not require you to rework code when deploying it. |

**See Also:**

- *Oracle Database Development Guide* for an overview of JDBC
- *Oracle Database 2 Day + Java Developer's Guide* and *Oracle Database JDBC Developer's Guide*

## SQLJ

**SQLJ** is an ANSI standard for embedding SQL statements in Java programs. You can use client-side SQLJ programs. In addition, you can combine SQLJ programs with JDBC.

**Note:**

Starting with Oracle Database 12c release 2 (12.2), Oracle Database does not support running *server-side* SQLJ code, including running stored procedures, functions, and triggers in the database environment.

SQLJ provides a simple, but powerful, way to develop client-side and middle-tier applications that access databases from Java. A developer writes a program using SQLJ and then uses the

SQLJ translator to translate embedded SQL to pure JDBC-based Java code. At run time, the program can communicate with multi-vendor databases using standard JDBC drivers.

The following example shows a simple SQLJ executable statement:

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

> **See Also:**
>
> *Oracle Database SQLJ Developer's Guide*

# Overview of Triggers

A database **trigger** is a compiled stored program unit, written in either PL/SQL or Java, that Oracle Database invokes ("fires") automatically in certain situations.

A trigger fires whenever one of the following operations occurs:

1. DML statements on a particular table or view, issued by any user

   DML statements modify data in schema objects. For example, inserting and deleting rows are DML operations.

2. DDL statements issued either by a particular user or any user

   DDL statements define schema objects. For example, creating a table and adding a column are DDL operations.

3. Database events

   User login or logoff, errors, and database startup or shutdown are events that can invoke triggers.

Triggers are schema objects that are similar to subprograms but differ in the way they are invoked. A subprogram is explicitly run by a user, application, or trigger. Triggers are implicitly invoked by the database when a triggering event occurs.

- Advantages of Triggers
  The correct use of triggers enables you to build and deploy applications that are more robust and that use the database more effectively.
- Types of Triggers
  Triggers can be categorized according to their means of invocation and the type of actions they perform.
- Timing for Triggers
  You can define the trigger timing—whether the trigger action is to be run before or after the triggering statement.
- Creation of Triggers
  The `CREATE TRIGGER` statement creates or replaces a database trigger.
- Execution of Triggers
  Oracle Database executes a trigger internally using the same steps as for subprogram execution.

- Storage of Triggers
  Oracle Database stores PL/SQL triggers in compiled form in a database schema, just like PL/SQL stored procedures.

> **See Also:**
>
> - "Overview of SQL Statements" to learn about DML and DDL
> - "Overview of Database Instance Startup and Shutdown"

## Advantages of Triggers

The correct use of triggers enables you to build and deploy applications that are more robust and that use the database more effectively.

You can use triggers to:

- Automatically generate derived column values
- Prevent invalid transactions
- Provide auditing and event logging
- Record information about table access

You can use triggers to enforce low-level business rules common for all client applications. For example, several applications may access the `employees` table. If a trigger on this table ensures the format of inserted data, then this business logic does not need to be reproduced in every client. Because the trigger cannot be circumvented by the application, the business logic in the trigger is used automatically.

You can use both triggers and integrity constraints to define and enforce any type of integrity rule. However, Oracle strongly recommends that you only use triggers to enforce complex business rules not definable using an integrity constraint.

Excessive use of triggers can result in complex interdependencies that can be difficult to maintain in a large application. For example, when a trigger is invoked, a SQL statement within its trigger action potentially can fire other triggers, resulting in cascading triggers that can produce unintended effects.

> **See Also:**
>
> - "Introduction to Data Integrity"
> - *Oracle Database 2 Day Developer's Guide*
> - *Oracle Database PL/SQL Language Reference* for guidelines and restrictions when planning triggers for your application

## Types of Triggers

Triggers can be categorized according to their means of invocation and the type of actions they perform.

Oracle Database supports the following types of triggers:

- Row triggers

  A row trigger fires each time the table is affected by the triggering statement. For example, if a statement updates multiple rows, then a row trigger fires once for each row affected by the `UPDATE`. If a triggering statement affects no rows, then a row trigger is not run. Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected.

- Statement triggers

  A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows affected by the triggering statement. For example, if a statement deletes 100 rows from a table, a statement-level `DELETE` trigger is fired only once. Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.

- `INSTEAD OF` triggers

  An INSTEAD OF trigger is fired by Oracle Database instead of executing the triggering statement. These triggers are useful for transparently modifying views that cannot be modified directly through DML statements.

- Event triggers

  You can use triggers to publish information about database events to subscribers. Event triggers are divided into the following categories:

  – A system event trigger can be caused by events such as database instance startup and shutdown or error messages.

  – A user event trigger is fired because of events related to user logon and logoff, DDL statements, and DML statements.

> **See Also:**
>
> - *Oracle Database 2 Day Developer's Guide*
> - *Oracle Database PL/SQL Language Reference*

## Timing for Triggers

You can define the trigger timing—whether the trigger action is to be run before or after the triggering statement.

A simple trigger is a single trigger on a table that enables you to specify actions for exactly one of the following timing points:

- Before the firing statement
- Before each row affected by the firing statement
- After each row affected by the firing statement
- After the firing statement

For statement and row triggers, a `BEFORE` trigger can enhance security and enable business rules before making changes to the database. The `AFTER` trigger is ideal for logging actions.

A compound trigger can fire at multiple timing points. Compound triggers help program an approach in which the actions that you implement for various timing points share common data.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* to learn about compound triggers

## Creation of Triggers

The `CREATE TRIGGER` statement creates or replaces a database trigger.

A PL/SQL trigger has the following general syntactic form:

```
CREATE TRIGGER trigger_name
  triggering_statement
  [trigger_restriction]
BEGIN
 triggered_action;
END;
```

A PL/SQL trigger has the following basic components:

- Trigger name

  The name must be unique among other trigger names in the same schema. For example, the name may be `part_reorder_trigger`.

- The trigger event or statement

  A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to be invoked. For example, a user updates a table.

- Trigger restriction

  A trigger restriction specifies a Boolean expression that must be `true` for the trigger to fire. For example, the trigger is not invoked unless the number of available parts is less than a present reorder amount.

- Triggered action

  A triggered action is the procedure that contains the SQL statements and code to be run when a triggering statement is issued and the trigger restriction evaluates to true. For example, a user inserts a row into a pending orders table.

- Example: CREATE TRIGGER Statement
  This example creates a trigger that fires when an `INSERT`, `UPDATE`, or `DELETE` statement executes on a line items table.
- Example: Invoking a Row-Level Trigger
  In this scenario, a customer initiates two orders and adds and removes line items from the orders.

> **See Also:**
>
> - *Oracle Database 2 Day Developer's Guide* and *Oracle Database PL/SQL Language Reference* to learn how to create triggers
> - *Oracle Database SQL Language Reference* to learn about the `CREATE TRIGGER` statement

## Example: CREATE TRIGGER Statement

This example creates a trigger that fires when an `INSERT`, `UPDATE`, or `DELETE` statement executes on a line items table.

Suppose that you create the `orders` and `lineitems` tables with the following statements. The `orders` table contains a row for each unique order, whereas the `lineitems` table contains a row for each item in an order.

```
CREATE TABLE orders
( order_id NUMBER PRIMARY KEY,
  /* other attributes */
  line_items_count NUMBER DEFAULT 0 );

CREATE TABLE lineitems
( order_id REFERENCES orders,
  seq_no   NUMBER,
  /* other attributes */
  CONSTRAINT lineitems PRIMARY KEY(order_id,seq_no) );
```

The following statement creates a sample trigger that automatically updates the `orders` table with the number of items in an order:

```
CREATE OR REPLACE TRIGGER lineitems_trigger
  AFTER INSERT OR UPDATE OR DELETE ON lineitems
  FOR EACH ROW
BEGIN
  IF (INSERTING OR UPDATING)
  THEN
    UPDATE orders SET line_items_count = NVL(line_items_count,0)+1
    WHERE order_id = :new.order_id;
  END IF;
  IF (DELETING OR UPDATING)
  THEN
    UPDATE orders SET line_items_count = NVL(line_items_count,0)-1
    WHERE order_id = :old.order_id;
  END IF;
END;
/
```

In `lineitems_trigger`, the triggering statement is an `INSERT`, `UPDATE`, or `DELETE` on the `lineitems` table. No triggering restriction exists. The trigger is invoked for each row changed. The trigger has access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified: the old value (`:old`), and the new value (`:new`). If a session updates or inserts rows in

`lineitems` for an order, then after the action the trigger calculates the number of items in this order and updates the `orders` table with the count.

## Example: Invoking a Row-Level Trigger

In this scenario, a customer initiates two orders and adds and removes line items from the orders.

The scenario is based on the trigger created in Example: CREATE TRIGGER Statement.

**Table 10-3 Row-Level Trigger Scenario**

| SQL Statement | Triggered SQL Statement | Description |
|---|---|---|
| `SQL> INSERT INTO orders`<br>`  (order_id) VALUES (78);`<br><br>`1 row created.` | | The customer creates an order with ID 78. At this point the customer has no items in the order.<br>Because no action is performed on the `lineitems` table, the trigger is not invoked. |
| `SQL> INSERT INTO orders`<br>`  (order_id) VALUES (92);`<br><br>`1 row created.` | | The customer creates a separate order with ID 92. At this point the customer has no items in the order.<br>Because no action is performed on the `lineitems` table, the trigger is not invoked. |
| `SQL> INSERT INTO lineitems`<br>`  (order_id, seq_no)`<br>`  VALUES (78,1);`<br><br>`1 row created.` | `UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(NULL,0)+1`<br>`  WHERE order_id = 78;` | The customer adds an item to order 78.<br>The `INSERT` invokes the trigger. The triggered statement increases the line item count for order 78 from 0 to 1. |
| `SQL> INSERT INTO lineitems`<br>`  (order_id, seq_no)`<br>`  VALUES (78,2);`<br><br>`1 row created.` | `UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(1,0)+1`<br>`  WHERE order_id = 78;` | The customer adds an additional item to order 78.<br>The `INSERT` invokes the trigger. The triggered statement increases the line item count for order 78 from 1 to 2. |
| `SQL> SELECT * FROM orders;`<br><br>`ORDER_ID LINE_ITEMS_COUNT`<br>`--------- ----------------`<br>`78 2`<br>`92 0` | | The customer queries the status of the two orders. Order 78 contains two items. Order 92 contains no items. |
| `SQL> SELECT * FROM lineitems;`<br><br>`ORDER_ID SEQ_NO`<br>`---------- ----------`<br>`78 1`<br>`78 2` | | The customer queries the status of the line items. Each item is uniquely identified by the order ID and the sequence number. |

**Table 10-3 (Cont.) Row-Level Trigger Scenario**

| SQL Statement | Triggered SQL Statement | Description |
|---|---|---|
| `SQL> UPDATE lineitems`<br>`  SET order_id = 92;`<br><br>`2 rows updated.` | `UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(NULL,0)+1`<br>`  WHERE order_id = 92;`<br><br>`UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(2,0)-1`<br>`  WHERE order_id = 78;`<br><br>`UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(1,0)+1`<br>`  WHERE order_id = 92;`<br><br>`UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(1,0)-1`<br>`  WHERE order_id = 78;` | The customer moves the line items that were in order 78 to order 92.<br>The `UPDATE` statement changes 2 rows in the `lineitems` tables, which invokes the trigger once for each row.<br>Each time the trigger is invoked, both `IF` conditions in the trigger are met. The first condition increments the count for order 92, whereas the second condition decreases the count for order 78. Thus, four total `UPDATE` statements are run. |
| `SQL> SELECT * FROM orders;`<br><br>`ORDER_ID LINE_ITEMS_COUNT`<br>`--------- ----------------`<br>`78 0`<br>`92 2` | | The customer queries the status of the two orders. The net effect is that the line item count for order 92 has increased from 0 to 2, whereas the count for order 78 has decreased from 2 to 0. |
| `SQL> SELECT * FROM lineitems;`<br><br>`ORDER_ID SEQ_NO`<br>`---------- ----------`<br>`92 1`<br>`92 2` | | The customer queries the status of the line items. Each item is uniquely identified by the order ID and the sequence number. |
| `SQL> DELETE FROM lineitems;`<br><br>`2 rows deleted.` | `UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(2,0)-1`<br>`  WHERE order_id = 92;`<br><br>`UPDATE orders`<br>`  SET line_items_count =`<br>`  NVL(1,0)-1`<br>`  WHERE order_id = 92;` | The customer now removes all line items from all orders.<br>The `DELETE` statement changes 2 rows in the `lineitems` tables, which invokes the trigger once for each row. For each trigger invocation, only one `IF` condition in the trigger is met. Each time the condition decreases the count for order 92 by 1. Thus, two total `UPDATE` statements are run. |

**Table 10-3 (Cont.) Row-Level Trigger Scenario**

| SQL Statement | Triggered SQL Statement | Description |
|---|---|---|
| SQL> SELECT * FROM orders;<br><br>ORDER_ID LINE_ITEMS_COUNT<br>--------- ----------------<br>78 0<br>92 0<br><br>SQL> SELECT * FROM lineitems;<br><br>no rows selected | | The customer queries the status of the two orders. Neither order contains line items.<br>The customer also queries the status of the line items. No items exist. |

## Execution of Triggers

Oracle Database executes a trigger internally using the same steps as for subprogram execution.

The only subtle difference is that a user account has the right to fire a trigger if it has the privilege to run the triggering statement. With this exception, the database validates and runs triggers the same way as stored subprograms.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* to learn more about trigger execution

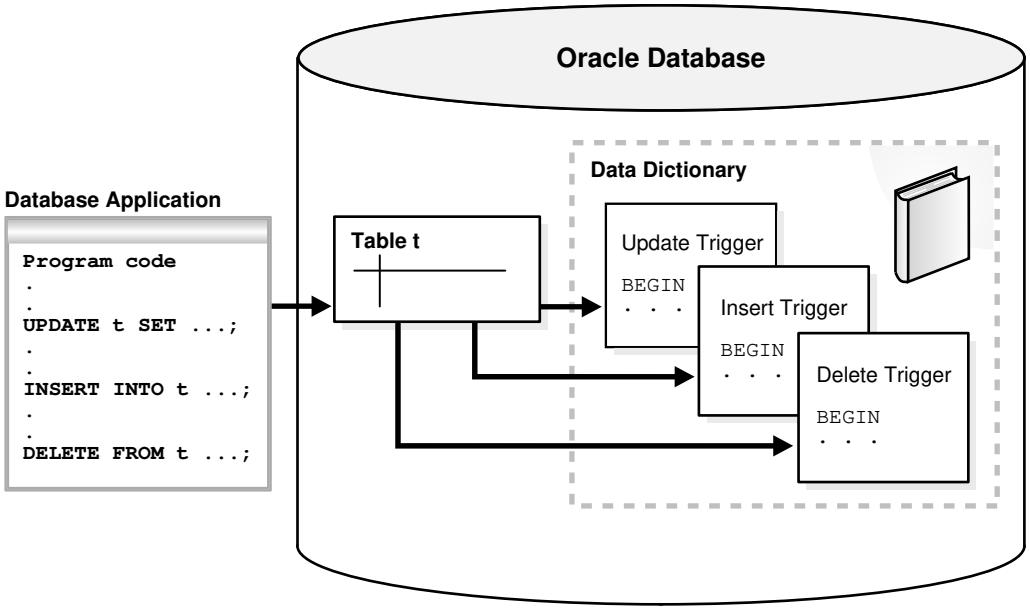
## Storage of Triggers

Oracle Database stores PL/SQL triggers in compiled form in a database schema, just like PL/SQL stored procedures.

When a `CREATE TRIGGER` statement commits, the compiled PL/SQL code is stored in the database. The shared pool removes the source code of the PL/SQL trigger.

The following graphic shows a database application with SQL statements that implicitly invoke PL/SQL triggers. The triggers are stored separately from their associated tables.

**Figure 10-7 Triggers**




Java triggers are stored in the same manner as PL/SQL triggers. However, a Java trigger references Java code that was separately compiled with a `CALL` statement. Thus, creating a Java trigger involves creating Java code and creating the trigger that references this Java code.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* to learn about compiling and storing triggers

# Part IV
# Oracle Transaction Management

Transaction management is the use of transactions to ensure data concurrency and consistency.

This part contains the following chapters:

- Data Concurrency and Consistency
- Transactions

- Data Concurrency and Consistency
  This chapter explains how Oracle Database maintains consistent data in a multiuser database environment.
- Transactions
  This chapter defines a transaction and describes how the database processes transactions.

11

# Data Concurrency and Consistency

This chapter explains how Oracle Database maintains consistent data in a multiuser database environment.

This chapter contains the following sections:


- Introduction to Data Concurrency and Consistency
- Overview of Oracle Database Transaction Isolation Levels
- Overview of the Oracle Database Locking Mechanism
- Overview of Automatic Locks
- Overview of Manual Data Locks
- Overview of User-Defined Locks


- Introduction to Data Concurrency and Consistency
  In a single-user database, a user can modify data without concern for other users modifying the same data at the same time. However, in a multiuser database, statements within multiple simultaneous transactions may update the same data. Transactions executing simultaneously must produce meaningful and consistent results.
- Overview of Oracle Database Transaction Isolation Levels
  The ANSI standard for transaction isolation levels is defined in terms of the phenomena that are either permitted or prevented for each isolation level.
- Overview of the Oracle Database Locking Mechanism
  A **lock** is a mechanism that prevents destructive interactions.
- Overview of Automatic Locks
  Oracle Database automatically locks a resource on behalf of a transaction to prevent other transactions from doing something that requires exclusive access to the same resource.
- Overview of Manual Data Locks
  You can manually override the Oracle Database default locking mechanisms.
- Overview of User-Defined Locks
  With Oracle Database Lock Management services, you can define your own locks for a specific application.

## Introduction to Data Concurrency and Consistency

In a single-user database, a user can modify data without concern for other users modifying the same data at the same time. However, in a multiuser database, statements within multiple simultaneous transactions may update the same data. Transactions executing simultaneously must produce meaningful and consistent results.

A multiuser database must provide the following:

- The assurance that users can access data at the same time (data concurrency)
- The assurance that each user sees a consistent view of the data (data consistency), including visible changes made by the user's own transactions and committed transactions of other users

To describe consistent transaction behavior when transactions run concurrently, database researchers have defined a transaction isolation model called serializability. A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.

While this degree of isolation between transactions is generally desirable, running many applications in serializable mode can seriously compromise application throughput. Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insertion into a table being queried by another transaction. In short, real-world considerations usually require a compromise between perfect transaction isolation and performance.

Oracle Database maintains data consistency by using a multiversion consistency model and various types of locks and transactions. In this way, the database can present a view of data to multiple concurrent users, with each view consistent to a point in time. Because different versions of data blocks can exist simultaneously, transactions can read the version of data committed at the point in time required by a query and return results that are consistent to a single point in time.

- Multiversion Read Consistency
  In Oracle Database, multiversioning is the ability to simultaneously materialize multiple versions of data. Oracle Database maintains multiversion read consistency.
- Locking Mechanisms
  In general, multiuser databases use some form of data locking to solve the problems associated with data concurrency, consistency, and integrity.
- ANSI/ISO Transaction Isolation Levels
  The SQL standard, which has been adopted by both ANSI and ISO/IEC, defines four levels of transaction isolation. These levels have differing degrees of impact on transaction processing throughput.

> **See Also:**
>
> "Data Integrity" and "Transactions "

## Multiversion Read Consistency

In Oracle Database, multiversioning is the ability to simultaneously materialize multiple versions of data. Oracle Database maintains multiversion read consistency.

Queries of an Oracle database have the following characteristics:

- Read-consistent queries

  The data returned by a query is committed and consistent for a single point in time.

> **Note:**
>
> Oracle Database *never* permits a dirty read, which occurs when a transaction reads uncommitted data in another transaction.

To illustrate the problem with dirty reads, suppose one transaction updates a column value without committing. A second transaction reads the updated and dirty (uncommitted) value. The first session rolls back the transaction so that the column has its old value, but the second transaction proceeds using the updated value, corrupting the database. Dirty reads compromise data integrity, violate foreign keys, and ignore unique constraints.

- Nonblocking queries

  Readers and writers of data do not block one another.

- Statement-Level Read Consistency
  Oracle Database always enforces **statement-level read consistency**, which guarantees that data returned by a single query is committed and consistent for a single point in time.
- Transaction-Level Read Consistency
  Oracle Database can also provide read consistency to all queries in a transaction, known as **transaction-level read consistency**.
- Read Consistency and Undo Segments
  To manage the multiversion read consistency model, the database must create a read-consistent set of data when a table is simultaneously queried and updated.
- Read Consistency and Deferred Inserts
  A special type of insert known as a **deferred insert** does not use the standard read consistency mechanism.

> **See Also:**
>
> "Summary of Locking Behavior"

## Statement-Level Read Consistency

Oracle Database always enforces **statement-level read consistency**, which guarantees that data returned by a single query is committed and consistent for a single point in time.

The point in time to which a single SQL statement is consistent depends on the transaction isolation level and the nature of the query:

- In the read committed isolation level, this point is the time at which the *statement* was opened. For example, if a `SELECT` statement opens at SCN 1000, then this statement is consistent to SCN 1000.
- In a serializable or read-only transaction, this point is the time the *transaction* began. For example, if a transaction begins at SCN 1000, and if multiple `SELECT` statements occur in this transaction, then each statement is consistent to SCN 1000.
- In a Flashback Query operation (`SELECT ... AS OF`), the `SELECT` statement explicitly specifies the point in time. For example, you can query a table as it appeared last Thursday at 2 p.m.

> **See Also:**
>
> *Oracle Database Development Guide* to learn about Flashback Query

## Transaction-Level Read Consistency

Oracle Database can also provide read consistency to all queries in a transaction, known as **transaction-level read consistency**.

In this case, each statement in a transaction sees data from the *same* point in time. This is the time at which the transaction began.

Queries made by a serializable transaction see changes made by the transaction itself. For example, a transaction that updates `employees` and then queries `employees` will see the updates. Transaction-level read consistency produces repeatable reads and does not expose a query to phantom reads.

## Read Consistency and Undo Segments

To manage the multiversion read consistency model, the database must create a read-consistent set of data when a table is simultaneously queried and updated.

Oracle Database achieves read consistency through undo data.

Whenever a user modifies data, Oracle Database creates undo entries, which it writes to undo segments. The undo segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Thus, multiple versions of the same data, all at different points in time, can exist in the database. The database can use snapshots of data at different points in time to provide read-consistent views of the data and enable nonblocking queries.

Read consistency is guaranteed in single-instance and Oracle Real Application Clusters (Oracle RAC) environments. Oracle RAC uses a cache-to-cache block transfer mechanism known as cache fusion to transfer read-consistent images of data blocks from one database instance to another.

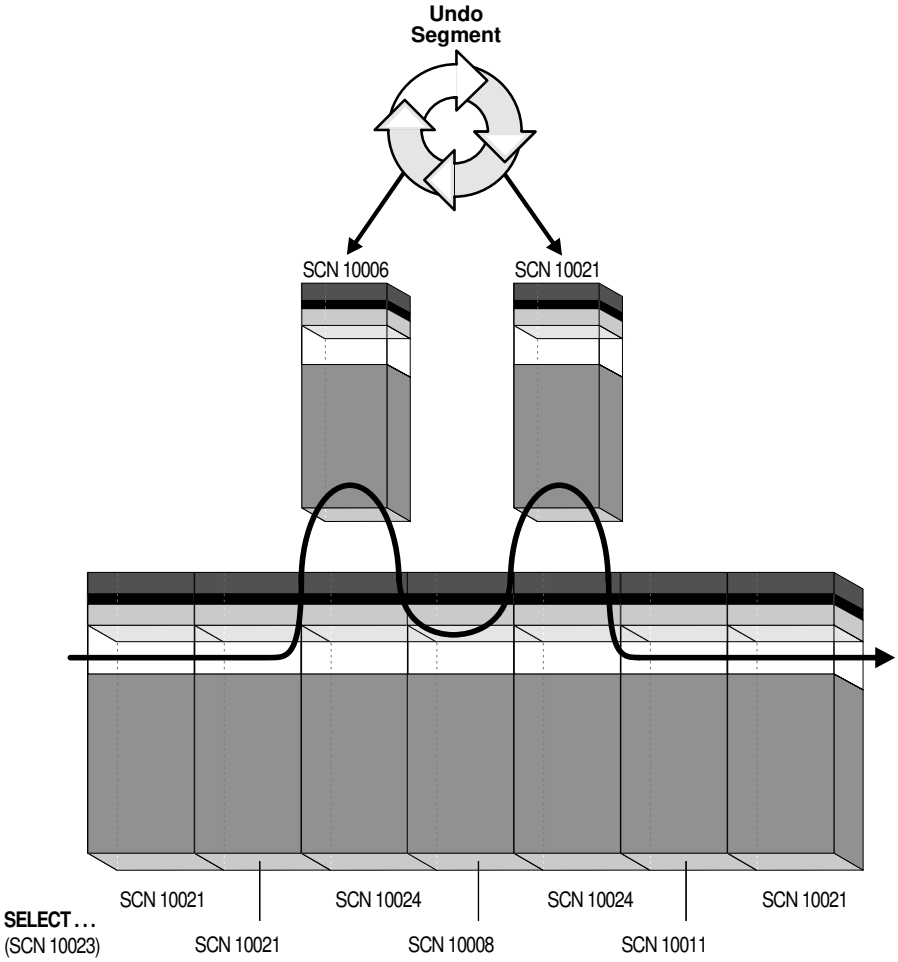
- Read Consistency: Example
  This example shows a query that uses undo data to provide statement-level read consistency in the read committed isolation level.
- Read Consistency and Interested Transaction Lists
  The **block header** of every segment block contains an **interested transaction list (ITL)**.

> **See Also:**
>
> - "Undo Segments" to learn about undo storage
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* to learn about read consistency mechanisms for LOBs

### Read Consistency: Example

This example shows a query that uses undo data to provide statement-level read consistency in the read committed isolation level.

**Figure 11-1 Read Consistency in the Read Committed Isolation Level**




As the database retrieves data blocks on behalf of a query, the database ensures that the data in each block reflects the contents of the block when the query began. The database rolls back changes to the block as needed to reconstruct the block to the point in time the query started processing.

The database uses an internal ordering mechanism called an SCN to guarantee the order of transactions. As the `SELECT` statement enters the execution phase, the database determines the SCN recorded at the time the query began executing. In Figure 11-1, this SCN is 10023. The query only sees committed data with respect to SCN 10023.

In Figure 11-1, blocks with SCNs *after* 10023 indicate changed data, as shown by the two blocks with SCN 10024. The `SELECT` statement requires a version of the block that is consistent with committed changes. The database copies current data blocks to a new buffer and applies undo data to reconstruct previous versions of the blocks. These reconstructed data blocks are called *consistent read (CR) clones*.

In Figure 11-1, the database creates two CR clones: one block consistent to SCN 10006 and the other block consistent to SCN 10021. The database returns the reconstructed data for the query. In this way, Oracle Database prevents dirty reads.

> **See Also:**
>
> "Database Buffer Cache" and "System Change Numbers (SCNs)"

### Read Consistency and Interested Transaction Lists

The **block header** of every segment block contains an **interested transaction list (ITL)**.

The database uses the ITL to determine whether a transaction was uncommitted when the database began modifying the block.

Entries in the ITL describe which transactions have rows locked and which rows in the block contain committed and uncommitted changes. The ITL points to the transaction table in the undo segment, which provides information about the timing of changes made to the database.

In a sense, the block header contains a recent history of transactions that affected each row in the block. The `INITRANS` parameter of the `CREATE TABLE` and `ALTER TABLE` statements controls the amount of transaction history that is kept.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about the `INITRANS` parameter

## Read Consistency and Deferred Inserts

A special type of insert known as a **deferred insert** does not use the standard read consistency mechanism.

A deferred insert uses the `MEMOPTIMIZE_WRITE` hint to insert into a table specified as `MEMOPTIMIZE FOR WRITE`. The database buffers these inserts in the large pool rather than in the buffer cache. The database does not track the changes using redo and undo. Instead, the database automatically commits the changes when the Space Management Coordinator (SMCO) writes the buffer to disk. The changes cannot be rolled back.

Deferred inserts differ from conventional inserts in important ways:

- Data residing in the large pool, which the application assumes is committed, can be lost. For example, the database instance could fail before the changes are saved to disk, even though the application reports that the changes are saved.
- Reading data directly from memory is not permitted. A writer cannot read its own changes until the background processes writes them to disk. No reader can see committed changes until they are written to disk.

Client applications that must avoid data loss should keep a local copy of data after writing to the large pool. The client can use the `DBMS_MEMOPTIMIZE` package to track the durability of the writes to memory, and the `DBMS_MEMOPTIMIZE_ADMIN` package to force the database to write to disk.

> **See Also:**
>
> - "Large Pool Buffers for Deferred Inserts"
> - *Oracle Database SQL Language Reference* to learn more about the `MEMOPTIMIZE FOR WRITE` hint

## Locking Mechanisms

In general, multiuser databases use some form of data locking to solve the problems associated with data concurrency, consistency, and integrity.

A lock is a mechanism that prevent destructive interaction between transactions accessing the same resource.

> **See Also:**
>
> "Overview of the Oracle Database Locking Mechanism"

## ANSI/ISO Transaction Isolation Levels

The SQL standard, which has been adopted by both ANSI and ISO/IEC, defines four levels of transaction isolation. These levels have differing degrees of impact on transaction processing throughput.

These isolation levels are defined in terms of phenomena that must be prevented between concurrently executing transactions. The preventable phenomena are:

- Dirty reads

  A transaction reads data that has been written by another transaction that has not been committed yet.

- Nonrepeatable (fuzzy) reads

  A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data. For example, a user queries a row and then later queries the same row, only to discover that the data has changed.

- Phantom reads

  A transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

  For example, a transaction queries the number of employees. Five minutes later it performs the same query, but now the number has increased by one because another user inserted a record for a new hire. More data satisfies the query criteria than before, but unlike in a fuzzy read the previously read data is unchanged.

The SQL standard defines four levels of isolation in terms of the phenomena that a transaction running at a particular isolation level is permitted to experience. Table 11-1 shows the levels.

**Table 11-1 Preventable Read Phenomena by Isolation Level**

| Isolation Level | Dirty Read | Nonrepeatable 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 |

Oracle Database offers the read committed (default) and serializable isolation levels. Also, the database offers a read-only mode.

**See Also:**

- "Overview of Oracle Database Transaction Isolation Levels" to learn about read committed, serializable, and read-only isolation levels
- *Oracle Database SQL Language Reference* for a discussion of Oracle Database conformance to SQL standards

# Overview of Oracle Database Transaction Isolation Levels

The ANSI standard for transaction isolation levels is defined in terms of the phenomena that are either permitted or prevented for each isolation level.

Oracle Database provides the transaction isolation levels:

- Read Committed Isolation Level
- Serializable Isolation Level
- Read-Only Isolation Level

- Read Committed Isolation Level
  In the **read committed isolation level**, every query executed by a transaction sees only data committed before the query—not the transaction—began.
- Serializable Isolation Level
  In the **serializable isolation level**, a transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself.
- Read-Only Isolation Level
  The **read-only isolation level** is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction unless the user is `SYS`.

> **See Also:**
>
> - *Oracle Database Development Guide* to learn more about transaction isolation levels
> - *Oracle Database SQL Language Reference* and *Oracle Database PL/SQL Language Reference* to learn about `SET TRANSACTION ISOLATION LEVEL`

## Read Committed Isolation Level

In the **read committed isolation level**, every query executed by a transaction sees only data committed before the query—not the transaction—began.

This isolation level is the default. It is appropriate for database environments in which few transactions are likely to conflict.

A query in a read committed transaction avoids reading data that commits while the query is in progress. For example, if a query is halfway through a scan of a million-row table, and if a different transaction commits an update to row 950,000, then the query does not see this change when it reads row 950,000. However, because the database does not prevent other transactions from modifying data read by a query, other transactions may change data *between* query executions. Thus, a transaction that runs the same query twice may experience fuzzy reads and phantoms.

- Read Consistency in the Read Committed Isolation Level
  The database provides a consistent result set for every query, guaranteeing data consistency, with no action by the user.
- Conflicting Writes in Read Committed Transactions
  In a read committed transaction, a **conflicting write** occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction.

## Read Consistency in the Read Committed Isolation Level

The database provides a consistent result set for every query, guaranteeing data consistency, with no action by the user.

An implicit query, such as a query implied by a `WHERE` clause in an `UPDATE` statement, is guaranteed a consistent set of results. However, each statement in an implicit query does not see the changes made by the DML statement itself, but sees the data as it existed before changes were made.

If a `SELECT` list contains a PL/SQL function, then the database applies statement-level read consistency at the statement level for SQL run within the PL/SQL function code, rather than at the parent SQL level. For example, a function could access a table whose data is changed and committed by another user. For each execution of the `SELECT` in the function, a new read-consistent snapshot is established.

> **See Also:**
>
> "Subqueries"

## Conflicting Writes in Read Committed Transactions

In a read committed transaction, a **conflicting write** occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction.

The transaction that prevents the row modification is sometimes called a **blocking transaction**. The read committed transaction waits for the blocking transaction to end and release its row lock.

The options are as follows:

- If the blocking transaction rolls back, then the waiting transaction proceeds to change the previously locked row as if the other transaction never existed.
- If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed row.

The following table shows how transaction 1, which can be either serializable or read committed, interacts with read committed transaction 2. It shows a classic situation known as a lost update. The update made by transaction 1 is not in the table *even though transaction 1 committed it*. Devising a strategy to handle lost updates is an important part of application development.

**Table 11-2 Conflicting Writes and Lost Updates in a READ COMMITTED Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 6200`<br>`Greene 9500` | No action. | Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found. |
| `SQL> UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';` | No action. | Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is `READ COMMITTED`. |
| No action. | `SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;` | Session 2 begins transaction 2 and sets the isolation level explicitly to `READ COMMITTED`. |
| No action. | `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 6200`<br>`Greene 9500` | Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1. |

**Table 11-2 (Cont.) Conflicting Writes and Lost Updates in a READ COMMITTED Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| No action. | `SQL> UPDATE employees`<br>`SET salary = 9900`<br>`WHERE last_name='Greene';` | Transaction 2 updates the salary for Greene successfully because transaction 1 locked only the Banda row. |
| `SQL> INSERT INTO`<br>`employees (employee_id,`<br>`last_name, email,`<br>`hire_date, job_id)`<br>`VALUES (210, 'Hintz',`<br>`'JHINTZ', SYSDATE,`<br>`'SH_CLERK');` | No action. | Transaction 1 inserts a row for employee Hintz, but does not commit. |
| No action. | `SQL> SELECT last_name,`<br>`salary FROM employees`<br>`WHERE last_name IN`<br>`('Banda', 'Greene',`<br>`'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 6200`<br>`Greene 9900` | Transaction 2 queries the salaries for employees Banda, Greene, and Hintz.<br>Transaction 2 sees its own update to the salary for Greene. Transaction 2 does not see the uncommitted update to the salary for Banda or the insertion for Hintz made by transaction 1. |
| No action. | `SQL> UPDATE employees SET`<br>`salary = 6300 WHERE`<br>`last_name = 'Banda';`<br><br>`-- prompt does not return` | Transaction 2 attempts to update the row for Banda, which is currently locked by transaction 1, creating a conflicting write. Transaction 2 waits until transaction 1 ends. |
| `SQL> COMMIT;` | No action. | Transaction 1 commits its work, ending the transaction. |
| No action. | `1 row updated.`<br><br>`SQL>` | The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary for Banda. |
| No action. | `SQL> SELECT last_name,`<br>`salary FROM employees`<br>`WHERE last_name IN`<br>`('Banda', 'Greene',`<br>`'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 6300`<br>`Greene 9900`<br>`Hintz` | Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. The Hintz insert committed by transaction 1 is now visible to transaction 2. Transaction 2 sees its own update to the Banda salary. |
| No action. | `COMMIT;` | Transaction 2 commits its work, ending the transaction. |

**Table 11-2 (Cont.) Conflicting Writes and Lost Updates in a READ COMMITTED Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`---------- ----------`<br>`Banda 6300`<br>`Greene 9900`<br>`Hintz` | No action. | Session 1 queries the rows for Banda, Greene, and Hintz. The salary for Banda is 6300, which is the update made by transaction 2. The update of Banda's salary to 7000 made by transaction 1 is now "lost." |

> **See Also:**
>
> - "Use of Locks" to learn about lost updates
> - "Row Locks (TX)" to learn when and why the database obtains row locks

## Serializable Isolation Level

In the **serializable isolation level**, a transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself.

A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database. Serializable isolation is suitable for environments:

- With large databases and short transactions that update only a few rows
- Where the chance that two concurrent transactions will modify the same rows is relatively low
- Where relatively long-running transactions are primarily read only

In serializable isolation, the read consistency normally obtained at the statement level extends to the entire transaction. Any row read by the transaction is assured to be the same when reread. Any query is guaranteed to return the same results for the duration of the transaction, so changes made by other transactions are not visible to the query regardless of how long it has been running. Serializable transactions do not experience dirty reads, fuzzy reads, or phantom reads.

Oracle Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were *already* committed when the serializable transaction began. The database generates an error when a serializable transaction tries to update or delete data changed by a different transaction that committed *after* the serializable transaction began:

```
ORA-08177: Cannot serialize access for this transaction
```

When a serializable transaction fails with the `ORA-08177` error, an application can take several actions, including the following:

- Commit the work executed to that point
- Execute additional (but different) statements, perhaps after rolling back to a savepoint established earlier in the transaction
- Roll back the entire transaction

The following table shows how a serializable transaction interacts with other transactions. If the serializable transaction does not try to change a row committed by another transaction after the serializable transaction began, then a serialized access problem is avoided.

**Table 11-3 Serializable Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 6200`<br>`Greene 9500` | No action. | Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found. |
| `SQL> UPDATE employees SET salary = 7000 WHERE last_name='Banda';` | No action. | Session 1 begins transaction 1 by updating the Banda salary. The default isolation level is `READ COMMITTED`. |
| No action. | `SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;` | Session 2 begins transaction 2 and sets it to the `SERIALIZABLE` isolation level. |
| No action. | `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------ ----------`<br>`Banda 6200`<br>`Greene 9500` | Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda *before* the uncommitted update made by transaction 1. |
| No action. | `SQL> UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';` | Transaction 2 updates the Greene salary successfully because only the Banda row is locked. |

**Table 11-3 (Cont.) Serializable Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| `SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE,'SH_CLERK');` | No action. | Transaction 1 inserts a row for employee Hintz. |
| `SQL> COMMIT;` | No action. | Transaction 1 commits its work, ending the transaction. |
| `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 7000`<br>`Greene 9500`<br>`Hintz` | `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ---------`<br>`Banda 6200`<br>`Greene 9900` | Session 1 queries the salaries for employees Banda, Greene, and Hintz and sees changes committed by transaction 1. Session 1 does not see the uncommitted Greene update made by transaction 2.<br>Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. Oracle Database read consistency ensures that the Hintz insert and Banda update committed by transaction 1 are *not* visible to transaction 2. Transaction 2 sees its own update to the Greene salary. |
| No action. | `COMMIT;` | Transaction 2 commits its work, ending the transaction. |
| `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ----------`<br>`Banda 7000`<br>`Greene 9900`<br>`Hintz` | `SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');`<br><br>`LAST_NAME SALARY`<br>`------------- ---------`<br>`Banda 7000`<br>`Greene 9900`<br>`Hintz` | Both sessions query the salaries for Banda, Greene, and Hintz. Each session sees all committed changes made by transaction 1 and transaction 2. |
| `SQL> UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz';` | No action. | Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is `READ COMMITTED`. |
| No action. | `SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;` | Session 2 begins transaction 4 and sets it to the `SERIALIZABLE` isolation level. |

**Table 11-3 (Cont.) Serializable Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| No action. | SQL> UPDATE employees<br>SET salary = 7200<br>WHERE last_name = 'Hintz';<br><br>-- prompt does not return | Transaction 4 attempts to update the salary for Hintz, but is blocked because transaction 3 locked the Hintz row. Transaction 4 queues behind transaction 3. |
| SQL> COMMIT; | No action. | Transaction 3 commits its update of the Hintz salary, ending the transaction. |
| No action. | UPDATE employees<br>SET salary = 7200<br>WHERE last_name = 'Hintz'<br>*<br>ERROR at line 1:<br>ORA-08177: can't<br>serialize access for<br>this transaction | The commit that ends transaction 3 causes the Hintz update in transaction 4 to fail with the ORA-08177 error. The problem error occurs because transaction 3 committed the Hintz update *after* transaction 4 began. |
| No action. | SQL> ROLLBACK; | Session 2 rolls back transaction 4, which ends the transaction. |
| No action. | SQL> SET TRANSACTION<br>ISOLATION LEVEL SERIALIZABLE; | Session 2 begins transaction 5 and sets it to the SERIALIZABLE isolation level. |
| No action. | SQL> SELECT last_name,<br>salary FROM employees<br>WHERE last_name IN<br>('Banda', 'Greene', 'Hintz');<br><br>LAST_NAME SALARY<br>------------- ----------<br>Banda 7000<br>Greene 9900<br>Hintz 7100 | Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible. |

**Table 11-3 (Cont.) Serializable Transaction**

| Session 1 | Session 2 | Explanation |
|---|---|---|
| No action. | `SQL> UPDATE employees`<br>`SET salary = 7200`<br>`WHERE last_name='Hintz';`<br><br>`1 row updated.` | Transaction 5 updates the Hintz salary to a different value. Because the Hintz update made by transaction 3 committed *before* the start of transaction 5, the serialized access problem is avoided.<br>**Note:** If a different transaction updated and committed the Hintz row after transaction 5 began, then the serialized access problem would occur again. |
| No action. | `SQL> COMMIT;` | Session 2 commits the update without any problems, ending the transaction. |

**See Also:**

- "Row Locks (TX)"
- "Overview of Transaction Control"

## Read-Only Isolation Level

The **read-only isolation level** is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction unless the user is `SYS`.

Read-only transactions are not susceptible to the `ORA-08177` error. Read-only transactions are useful for generating reports in which the contents must be consistent with respect to the time when the transaction began.

Oracle Database achieves read consistency by reconstructing data as needed from the undo segments. Because undo segments are used in a circular fashion, the database can overwrite undo data. Long-running reports run the risk that undo data required for read consistency may have been reused by a different transaction, raising a `snapshot too old` error. Setting an undo retention period, which is the minimum amount of time that the database attempts to retain old undo data before overwriting it, appropriately avoids this problem.

**See Also:**

- "Undo Segments"
- *Oracle Database Administrator's Guide* to learn how to set the undo retention period

# Overview of the Oracle Database Locking Mechanism

A **lock** is a mechanism that prevents destructive interactions.

Interactions are destructive when they incorrectly update data or incorrectly alter underlying data structures, between transactions accessing shared data. Locks play a crucial role in maintaining database concurrency and consistency.

- Summary of Locking Behavior
  The database maintains several different types of locks, depending on the operation that acquired the lock.
- Use of Locks
  In a single-user database, locks are not necessary because only one user is modifying information. However, when multiple users are accessing and modifying data, the database must provide a way to prevent concurrent modification of the same data.
- Lock Modes
  Oracle Database automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity.
- Lock Conversion and Escalation
  Oracle Database performs **lock conversion** as necessary.
- Lock Duration
  Oracle Database automatically releases a lock when some event occurs so that the transaction no longer requires the resource.
- Locks and Deadlocks
  A **deadlock** is a situation in which two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work.

## Summary of Locking Behavior

The database maintains several different types of locks, depending on the operation that acquired the lock.

In general, the database uses two types of locks: exclusive locks and share locks. Only one exclusive lock can be obtained on a resource such as a row or a table, but many share locks can be obtained on a single resource.

Locks affect the interaction of readers and writers. A reader is a query of a resource, whereas a writer is a statement modifying a resource. The following rules summarize the locking behavior of Oracle Database for readers and writers:

- A row is locked only when modified by a writer.

  When a statement updates one row, the transaction acquires a lock for this row only. By locking table data at the row level, the database minimizes contention for the same data. Under normal circumstances[1] the database does not escalate a row lock to the block or table level.
- A writer of a row blocks a concurrent writer of the same row.

  If one transaction is modifying a row, then a row lock prevents a different transaction from modifying the same row simultaneously.

[1] When processing a distributed two-phase commit, the database may briefly prevent read access in special circumstances. Specifically, if a query starts between the prepare and commit phases and attempts to read the data before the commit, then the database may escalate a lock from row-level to block-level to guarantee read consistency.

- A reader never blocks a writer.

  Because a reader of a row does not lock it, a writer can modify this row. The only exception is a `SELECT ... FOR UPDATE` statement, which is a special type of `SELECT` statement that *does* lock the row that it is reading.

- A writer never blocks a reader.

  When a row is being changed by a writer, the database uses undo data to provide readers with a consistent view of the row.

> **Note:**
>
> Readers of data may have to wait for writers of the same data blocks in very special cases of pending distributed transactions.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* to learn about `SELECT ... FOR UPDATE`
> - *Oracle Database Administrator's Guide* to learn about waits associated with in-doubt distributed transactions

## Use of Locks

In a single-user database, locks are not necessary because only one user is modifying information. However, when multiple users are accessing and modifying data, the database must provide a way to prevent concurrent modification of the same data.

Locks achieve the following important database requirements:

- Consistency

  The data a session is viewing or changing must not be changed by other sessions until the user is finished.

- Integrity

  The data and structures must reflect all changes made to them in the correct sequence.

Oracle Database provides data concurrency, consistency, and integrity among transactions through its locking mechanisms. Locking occurs automatically and requires no user action.

The need for locks can be illustrated by a concurrent update of a single row. In the following example, a simple web-based application presents the end user with an employee email and phone number. The application uses an `UPDATE` statement such as the following to modify the data:

```
UPDATE employees
SET    email = ?, phone_number = ?
WHERE  employee_id = ?
AND    email = ?
AND    phone_number = ?
```

In the preceding `UPDATE` statement, the email and phone number values in the `WHERE` clause are the original, unmodified values for the specified employee. This update ensures that the row that the application modifies was not changed after the application last read and displayed it to the user. In this way, the application avoids the lost update problem in which one user overwrites changes made by another user, effectively losing the update by the second user (Table 11-2 shows an example of a lost update).

Table 11-4 shows the sequence of events when two sessions attempt to modify the same row in the `employees` table at roughly the same time.

**Table 11-4 Row Locking Example**

| T | Session 1 | Session 2 | Description |
|---|---|---|---|
| t0 | `SELECT employee_id as ID,`<br>`email, phone_number`<br>`FROM hr.employees`<br>`WHERE last_name='Himuro';`<br><br>`ID EMAIL PHONE_NUMBER`<br>`--- ------- ------------`<br>`118 GHIMURO 515.127.4565` | | In session 1, the `hr1` user queries `hr.employees` for the Himuro record and displays the employee_id (`118`), email (`GHIMURO`), and phone number (`515.127.4565`) attributes. |
| t1 | | `SELECT employee_id as ID,`<br>`email, phone_number`<br>`FROM hr.employees`<br>`WHERE last_name='Himuro';`<br><br>`ID EMAIL PHONE_NUMBER`<br>`--- ------- ------------`<br>`118 GHIMURO 515.127.4565` | In session 2, the `hr2` user queries `hr.employees` for the Himuro record and displays the employee_id (`118`), email (`GHIMURO`), and phone number (`515.127.4565`) attributes. |
| t2 | `UPDATE hr.employees SET`<br>`phone_number='515.555.1234'`<br>`WHERE employee_id=118`<br>`AND email='GHIMURO'`<br>`AND phone_number =`<br>`'515.127.4565';`<br><br>`1 row updated.` | | In session 1, the `hr1` user updates the phone number in the row to `515.555.1234`, which acquires a lock on the `GHIMURO` row. |
| t3 | | `UPDATE hr.employees SET`<br>`phone_number='515.555.1235'`<br>`WHERE employee_id=118`<br>`AND email='GHIMURO'`<br>`AND phone_number =`<br>`'515.127.4565';`<br><br>`-- SQL*Plus does not show`<br>`-- a row updated message or`<br>`-- return the prompt.` | In session 2, the `hr2` user attempts to update the same row, but is blocked because `hr1` is currently processing the row.<br>The attempted update by `hr2` occurs almost simultaneously with the `hr1` update. |

**Table 11-4 (Cont.) Row Locking Example**

| T | Session 1 | Session 2 | Description |
|---|---|---|---|
| t4 | `COMMIT;`<br><br>`Commit complete.` | | In session 1, the `hr1` user commits the transaction.<br>The commit makes the change for Himuro permanent and unblocks session 2, which has been waiting. |
| t5 | | `0 rows updated.` | In session 2, the `hr2` user discovers that the `GHIMURO` row was modified in such a way that it no longer matches its predicate.<br>Because the predicates do not match, session 2 updates no records. |
| t6 | `UPDATE hr.employees SET`<br>`phone_number='515.555.1235'`<br>`WHERE employee_id=118`<br>`AND email='GHIMURO' AND`<br>`phone_number='515.555.1234';`<br><br>`1 row updated.` | | In session 1, the `hr1` user realizes that it updated the `GHIMURO` row with the wrong phone number. The user starts a new transaction and updates the phone number in the row to `515.555.1235`, which locks the `GHIMURO` row. |
| t7 | | `SELECT employee_id as ID,`<br>`email, phone_number`<br>`FROM hr.employees`<br>`WHERE last_name='Himuro';`<br><br><br>`ID EMAIL PHONE_NUMBER`<br>`--- ------- ------------`<br>`118 GHIMURO 515.555.1234` | In session 2, the `hr2` user queries `hr.employees` for the Himuro record. The record shows the phone number update committed by session 1 at t4. Oracle Database read consistency ensures that session 2 does not see the uncommitted change made at t6. |
| t8 | | `UPDATE hr.employees SET`<br>`phone_number='515.555.1235`<br>`WHERE employee_id=118`<br>`AND email='GHIMURO'`<br>`AND phone_number =`<br>`'515.555.1234';`<br><br>`-- SQL*Plus does not show`<br>`-- a row updated message or`<br>`-- return the prompt.` | In session 2, the `hr2` user attempts to update the same row, but is blocked because `hr1` is currently processing the row. |
| t9 | `ROLLBACK;`<br><br>`Rollback complete.` | | In session 1, the `hr1` user rolls back the transaction, which ends it. |

**Table 11-4 (Cont.) Row Locking Example**

| T | Session 1 | Session 2 | Description |
|---|---|---|---|
| t10 | | `1 row updated.` | In session 2, the update of the phone number succeeds because the session 1 update was rolled back. The `GHIMURO` row matches its predicate, so the update succeeds. |
| t11 | | `COMMIT;`<br><br>`Commit complete.` | Session 2 commits the update, ending the transaction. |

Oracle Database automatically obtains necessary locks when executing SQL statements. For example, before the database permits a session to modify data, the session must first lock the data. The lock gives the session exclusive control over the data so that no other transaction can modify the locked data until the lock is released.

Because the locking mechanisms of Oracle Database are tied closely to transaction control, application designers need only define transactions properly, and Oracle Database automatically manages locking. Users never need to lock any resource explicitly, although Oracle Database also enables users to lock data manually.

The following sections explain concepts that are important for understanding how Oracle Database achieves data concurrency.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `OWA_OPT_LOCK` package, which contains subprograms that can help prevent lost updates

## Lock Modes

Oracle Database automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity.

The less restrictive the level, the more available the data is for access by other users. Conversely, the more restrictive the level, the more limited other transactions are in the types of locks that they can acquire.

Oracle Database uses two modes of locking in a multiuser database:

- Exclusive lock mode

  This mode prevents the associated resource from being shared. A transaction obtains an exclusive lock when it modifies data. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.

- Share lock mode

This mode allows the associated resource to be shared, depending on the operations involved. Multiple users reading data can share the data, each holding a share lock to prevent concurrent access by a writer who needs an exclusive lock. Multiple transactions can acquire share locks on the same resource.

Assume that a transaction uses a `SELECT ... FOR UPDATE` statement to select a single table row. The transaction acquires an exclusive row lock and a row share table lock. The row lock allows other sessions to modify any rows *other than* the locked row, while the table lock prevents sessions from altering the structure of the table. Thus, the database permits as many statements as possible to execute.

## Lock Conversion and Escalation

Oracle Database performs **lock conversion** as necessary.

In lock conversion, the database automatically converts a table lock of lower restrictiveness to one of higher restrictiveness. For example, suppose a transaction issues a `SELECT ... FOR UPDATE` for an employee and later updates the locked row. In this case, the database automatically converts the row share table lock to a row exclusive table lock. A transaction holds exclusive row locks for all rows inserted, updated, or deleted within the transaction. Because row locks are acquired at the highest degree of restrictiveness, no lock conversion is required or performed.

Lock conversion is different from lock escalation, which occurs when numerous locks are held at one level of granularity (for example, rows) and a database raises the locks to a higher level of granularity (for example, table). If a session locks many rows in a table, then some databases automatically escalate the row locks to a single table. The number of locks decreases, but the restrictiveness of what is locked increases.

*Oracle Database never escalates locks.* Lock escalation greatly increases the probability of deadlocks. Assume that a system is trying to escalate locks on behalf of transaction 1 but cannot because of the locks held by transaction 2. A deadlock is created if transaction 2 also requires lock escalation of the same data before it can proceed.

## Lock Duration

Oracle Database automatically releases a lock when some event occurs so that the transaction no longer requires the resource.

Usually, the database holds locks acquired by statements within a transaction for the duration of the transaction. These locks prevent destructive interference such as dirty reads, lost updates, and destructive DDL from concurrent transactions.

> **Note:**
>
> A table lock taken on a child table because of an unindexed foreign key is held for the duration of the statement, not the transaction. Also, the `DBMS_LOCK` package enables user-defined locks to be released and allocated at will and even held over transaction boundaries.

Oracle Database releases all locks acquired by the statements within a transaction when it commits or rolls back. Oracle Database also releases locks acquired after a savepoint when rolling back to the savepoint. However, only transactions not waiting for the previously locked

resources can acquire locks on the now available resources. Waiting transactions continue to wait until after the original transaction commits or rolls back completely.

> **See Also:**
>
> - "Table 12-3" that shows transaction waiting behavior
> - "Overview of User-Defined Locks" to learn more about `DBMS_LOCK`

## Locks and Deadlocks

A **deadlock** is a situation in which two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work.

Oracle Database automatically detects deadlocks and resolves them by rolling back one statement involved in the deadlock, releasing one set of the conflicting row locks. The database returns a corresponding message to the transaction that undergoes statement-level rollback. The statement rolled back belongs to the transaction that detects the deadlock. Usually, the signaled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Table 11-5 illustrates two transactions in a deadlock.

**Table 11-5 Deadlocked Transactions**

| T | Session 1 | Session 2 | Explanation |
|---|---|---|---|
| t0 | `SQL> UPDATE employees`<br>`  SET salary = salary*1.1`<br>`  WHERE employee_id = 100;`<br><br>`1 row updated.` | `SQL> UPDATE employees`<br>`  SET  salary = salary*1.1`<br>`  WHERE employee_id = 200;`<br><br>`1 row updated.` | Session 1 starts transaction 1 and updates the salary for employee 100. Session 2 starts transaction 2 and updates the salary for employee 200. No problem exists because each transaction locks only the row that it attempts to update. |
| t1 | `SQL> UPDATE employees`<br>`  SET salary = salary*1.1`<br>`  WHERE employee_id = 200;`<br><br>`-- prompt does not return` | `SQL> UPDATE employees`<br>`  SET salary = salary*1.1`<br>`  WHERE employee_id = 100;`<br><br>`-- prompt does not return` | Transaction 1 attempts to update the employee 200 row, which is currently locked by transaction 2. Transaction 2 attempts to update the employee 100 row, which is currently locked by transaction 1.<br>A deadlock results because neither transaction can obtain the resource it needs to proceed or terminate. No matter how long each transaction waits, the conflicting locks are held. |

**Table 11-5 (Cont.) Deadlocked Transactions**

| T | Session 1 | Session 2 | Explanation |
|---|---|---|---|
| t2 | `UPDATE employees`<br>`*`<br>`ERROR at line 1:`<br>`ORA-00060: deadlock detected`<br>`while waiting for resource`<br><br>`SQL>` | | Transaction 1 signals the deadlock and rolls back the `UPDATE` statement issued at t1. However, the update made at t0 is not rolled back. The prompt is returned in session 1.<br>**Note:** Only one session in the deadlock actually gets the deadlock error, but either session could get the error. |
| t3 | `SQL> COMMIT;`<br><br>`Commit complete.` | | Session 1 commits the update made at t0, ending transaction 1. The update unsuccessfully attempted at t1 is not committed. |
| t4 | | `1 row updated.`<br><br>`SQL>` | The update at t1 in transaction 2, which was being blocked by transaction 1, is executed. The prompt is returned. |
| t5 | | `SQL> COMMIT;`<br><br>`Commit complete.` | Session 2 commits the updates made at t0 and t1, which ends transaction 2. |

Deadlocks most often occur when transactions explicitly override the default locking of Oracle Database. Because Oracle Database does not escalate locks and does not use read locks for queries, but does use row-level (rather than page-level) locking, deadlocks occur infrequently.

**See Also:**

- "Overview of Manual Data Locks"
- *Oracle Database Development Guide* to learn how to handle deadlocks when you lock tables explicitly

# Overview of Automatic Locks

Oracle Database automatically locks a resource on behalf of a transaction to prevent other transactions from doing something that requires exclusive access to the same resource.

The database automatically acquires different types of locks at different levels of restrictiveness depending on the resource and the operation being performed.

> **Note:**
>
> The database never locks rows when performing simple reads.

Oracle Database locks are divided into the categories show in the following table.

**Table 11-6 Lock Categories**

| Lock | Description | To Learn More |
|---|---|---|
| DML Locks | Protect data. For example, table locks lock entire tables, while row locks lock selected rows. | "DML Locks" |
| DDL Locks | Protect the structure of schema objects—for example, the dictionary definitions of tables and views. | "DDL Locks" |
| System Locks | Protect internal database structures such as data files. Latches, mutexes, and internal locks are entirely automatic. | "System Locks" |

- DML Locks
  A **DML lock**, also called a *data lock*, guarantees the integrity of data accessed concurrently by multiple users.
- DDL Locks
  A **data dictionary (DDL) lock** protects the definition of a schema object while an ongoing DDL operation acts on or refers to the object.
- System Locks
  Oracle Database uses various types of system locks to protect internal database and memory structures. These mechanisms are inaccessible to users because users have no control over their occurrence or duration.

## DML Locks

A **DML lock**, also called a *data lock*, guarantees the integrity of data accessed concurrently by multiple users.

For example, a DML lock prevents two customers from buying the last copy of a book available from an online bookseller. DML locks prevent destructive interference of simultaneous conflicting DML or DDL operations.

DML statements automatically acquire the following types of locks:

- Row Locks (TX)
- Table Locks (TM)

In the following sections, the acronym in parentheses after each type of lock or lock mode is the abbreviation used in the Locks Monitor of Oracle Enterprise Manager (Enterprise Manager). Enterprise Manager might display TM for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

- Row Locks (TX)
  A **row lock**, also called a *TX lock*, is a lock on a single row of table. A transaction acquires a row lock for each row modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or `SELECT ... FOR UPDATE` statement. The row lock exists until the transaction commits or rolls back.
- Table Locks (TM)
  A **table lock**, also called a *TM lock*, is acquired by a transaction when a table is modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `SELECT` with the `FOR UPDATE` clause, or `LOCK TABLE` statement.
- Locks and Foreign Keys
  Oracle Database maximizes the concurrency control of parent keys in relation to dependent foreign keys.

## Row Locks (TX)

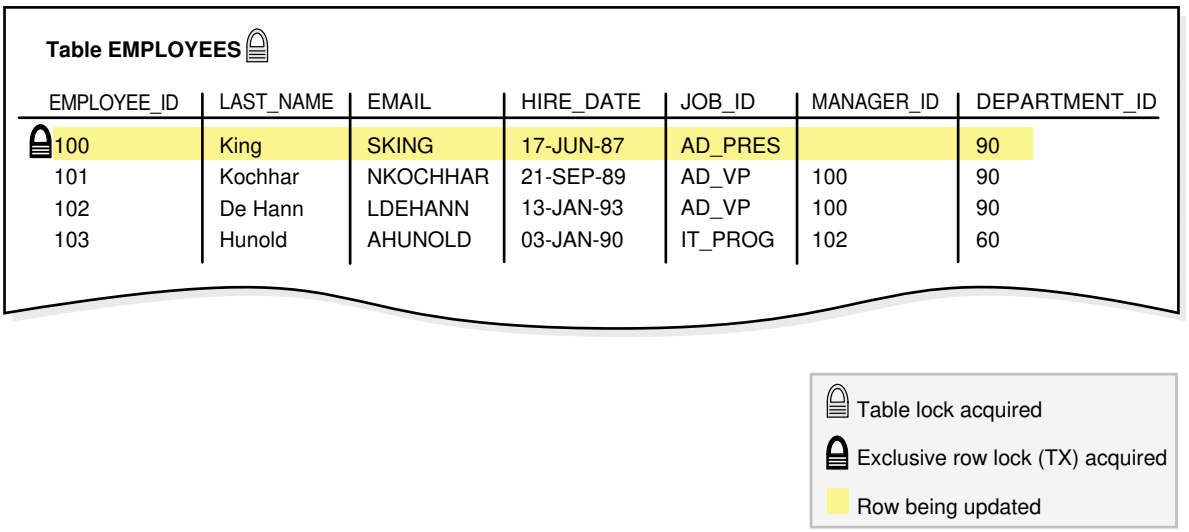
A **row lock**, also called a *TX lock*, is a lock on a single row of table. A transaction acquires a row lock for each row modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or `SELECT ... FOR UPDATE` statement. The row lock exists until the transaction commits or rolls back.

Row locks primarily serve as a queuing mechanism to prevent two transactions from modifying the same row. The database always locks a modified row in exclusive mode so that other transactions cannot modify the row until the transaction holding the lock commits or rolls back. Row locking provides the finest grain locking possible and so provides the best possible concurrency and throughput.

> **Note:**
>
> If a transaction terminates because of database instance failure, then block-level recovery makes a row available before the entire transaction is recovered.

If a transaction obtains a lock for a row, then the transaction also acquires a lock for the table containing the row. The table lock prevents conflicting DDL operations that would override data changes in a current transaction. Figure 11-2 illustrates an update of the third row in a table. Oracle Database automatically places an exclusive lock on the updated row and a subexclusive lock on the table.

**Figure 11-2 Row and Table Locks**




- Row Locks and Concurrency
  This scenario illustrates how Oracle Database uses row locks for concurrency.
- Storage of Row Locks
  Unlike some databases, which use a lock manager to maintain a list of locks in memory, Oracle Database stores lock information in the data block that contains the locked row.

## Row Locks and Concurrency

This scenario illustrates how Oracle Database uses row locks for concurrency.

Three sessions query the same rows simultaneously. Session 1 and 2 proceed to make uncommitted updates to different rows, while session 3 makes no updates. Each session sees its own uncommitted updates but not the uncommitted updates of any other session.

**Table 11-7 Data Concurrency Example**

| T | Session 1 | Session 2 | Session 3 | Explanation |
|---|---|---|---|---|
| t0 | `SELECT employee_id, salary FROM employees WHERE employee_id IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 600` | `SELECT employee_id, salary FROM employees WHERE employee_id IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 600` | `SELECT employee_id, salary FROM employees WHERE employee_id IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 600` | Three different sessions simultaneously query the ID and salary of employees 100 and 101. The results returned by each query are identical. |

**Table 11-7 (Cont.) Data Concurrency Example**

| T | Session 1 | Session 2 | Session 3 | Explanation |
|---|---|---|---|---|
| t1 | `UPDATE hr.employees`<br>`SET salary =`<br>`salary+100`<br>`WHERE`<br>`employee_id=100;` | | | Session 1 updates the salary of employee 100, but does not commit. In the update, the writer acquires a row-level lock for the updated row only, thereby preventing other writers from modifying this row. |
| t2 | `SELECT employee_id,`<br>`salary`<br>`FROM employees`<br>`WHERE employee_id`<br>`IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 612`<br>`101 600` | `SELECT employee_id,`<br>`salary`<br>`FROM employees`<br>`WHERE employee_id`<br>`IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 600` | `SELECT employee_id,`<br>`salary`<br>`FROM employees`<br>`WHERE employee_id`<br>`IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 600` | Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update. The readers in session 2 and 3 return rows immediately and do not wait for session 1 to end its transaction. The database uses multiversion read consistency to show the salary as it existed before the update in session 1. |
| t3 | | `UPDATE hr.employee`<br>`SET salary =`<br>`salary+100`<br>`WHERE`<br>`employee_id=101;` | | Session 2 updates the salary of employee 101, but does not commit the transaction. In the update, the writer acquires a row-level lock for the updated row only, preventing other writers from modifying this row. |
| t4 | `SELECT employee_id,`<br>`salary`<br>`FROM employees`<br>`WHERE employee_id`<br>`IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 612`<br>`101 600` | `SELECT employee_id,`<br>`salary`<br>`FROM employees`<br>`WHERE employee_id`<br>`IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 700` | `SELECT employee_id,`<br>`salary`<br>`FROM employees`<br>`WHERE employee_id`<br>`IN ( 100, 101 );`<br><br>`EMPLOYEE_ID SALARY`<br>`----------- ------`<br>`100 512`<br>`101 600` | Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update, but not the salary update for employee 101 made in session 2. The reader in session 2 shows the salary update made in session 2, but not the salary update made in session 1. The reader in session 3 uses read consistency to show the salaries before modification by session 1 and 2. |

> **See Also:**
>
> - *Oracle Database SQL Language Reference*
> - *Oracle Database Reference* to learn about `V$LOCK`

### Storage of Row Locks

Unlike some databases, which use a lock manager to maintain a list of locks in memory, Oracle Database stores lock information in the data block that contains the locked row.

The database uses a queuing mechanism for acquisition of row locks. If a transaction requires a lock for an unlocked row, then the transaction places a lock in the data block. Each row modified by this transaction points to a copy of the transaction ID stored in the block header.

When a transaction ends, the transaction ID remains in the block header. If a different transaction wants to modify a row, then it uses the transaction ID to determine whether the lock is active. If the lock is active, then the session asks to be notified when the lock is released. Otherwise, the transaction acquires the lock.

> **See Also:**
>
> - "Overview of Data Blocks" to learn more about the data block header
> - *Oracle Database Reference* to learn about `V$TRANSACTION`

## Table Locks (TM)

A **table lock**, also called a *TM lock*, is acquired by a transaction when a table is modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `SELECT` with the `FOR UPDATE` clause, or `LOCK TABLE` statement.

DML operations require table locks to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction.

A table lock can be held in any of the following modes:

- Row Share (RS)

  This lock, also called a *subshare table lock (SS)*, indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them. A row share lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.

- Row Exclusive Table Lock (RX)

  This lock, also called a *subexclusive table lock (SX)*, generally indicates that the transaction holding the lock has updated table rows or issued `SELECT ... FOR UPDATE`. An SX lock allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, SX locks allow multiple transactions to obtain simultaneous SX and subshare table locks for the same table.

- Share Table Lock (S)

A share table lock held by a transaction allows other transactions to query the table (without using `SELECT ... FOR UPDATE`), but updates are allowed only if a single transaction holds the share table lock. Because multiple transactions may hold a share table lock concurrently, holding this lock is not sufficient to ensure that a transaction can modify the table.

- Share Row Exclusive Table Lock (SRX)

  This lock, also called a *share-subexclusive table lock (SSX)*, is more restrictive than a share table lock. Only one transaction at a time can acquire an SSX lock on a given table. An SSX lock held by a transaction allows other transactions to query the table (except for `SELECT ... FOR UPDATE`) but not to update the table.

- Exclusive Table Lock (X)

  This lock is the most restrictive, prohibiting other transactions from performing any type of DML statement or placing any type of lock on the table.

**See Also:**

- *Oracle Database SQL Language Reference*
- *Oracle Database Development Guide* to learn more about table locks

## Locks and Foreign Keys

Oracle Database maximizes the concurrency control of parent keys in relation to dependent foreign keys.

Locking behavior depends on whether foreign key columns are indexed. If foreign keys are not indexed, then the child table will probably be locked more frequently, deadlocks will occur, and concurrency will be decreased. For this reason foreign keys should almost always be indexed. The only exception is when the matching unique or primary key is never updated or deleted.

- Locks and Unindexed Foreign Keys
  The database acquires a full table lock on the child table when no index exists on the foreign key column of the child table, and a session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.
- Locks and Indexed Foreign Keys
  The database does *not* acquire a full table lock on the child table when a foreign key column in the child table *is* indexed, and a session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

### Locks and Unindexed Foreign Keys

The database acquires a full table lock on the child table when no index exists on the foreign key column of the child table, and a session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

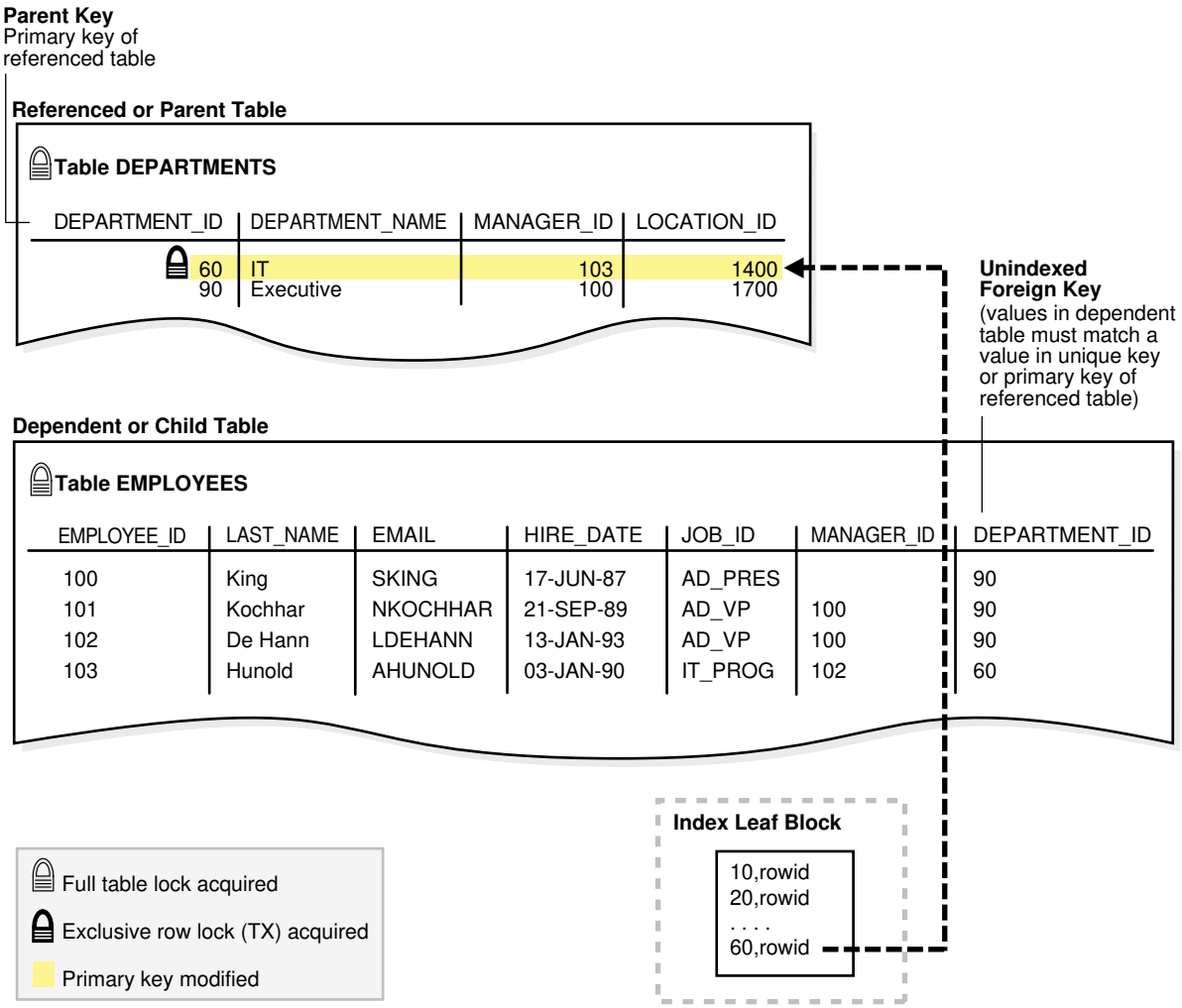
When both of the following conditions are true, the database acquires a full table lock on the child table:

- No index exists on the foreign key column of the child table.
- A session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

> **Note:**
>
> Inserts into the parent table do *not* acquire blocking table locks that prevent DML on the child table. In the case of inserts, the database acquires a lock on the child table that prevents structural changes, but not modifications of existing or newly added rows.

Suppose that `hr.departments` table is a parent of `hr.employees`, which contains the unindexed foreign key `employees.department_id`. The following figure shows a session modifying the primary key attributes of department 60 in the `departments` table.

**Figure 11-3 Locking Mechanisms with Unindexed Foreign Key**




In Figure 11-3, the database acquires a full table lock on `employees` during the primary key modification of department 60. This lock enables other sessions to query but not update the

`employees` table. For example, sessions cannot update employee phone numbers. The table lock on `employees` releases immediately after the primary key modification on the `departments` table completes. If multiple rows in `departments` undergo primary key modifications, then a table lock on `employees` is obtained and released once for each row that is modified in `departments`.

> **Note:**
>
> DML on a child table does not acquire a table lock on the parent table.

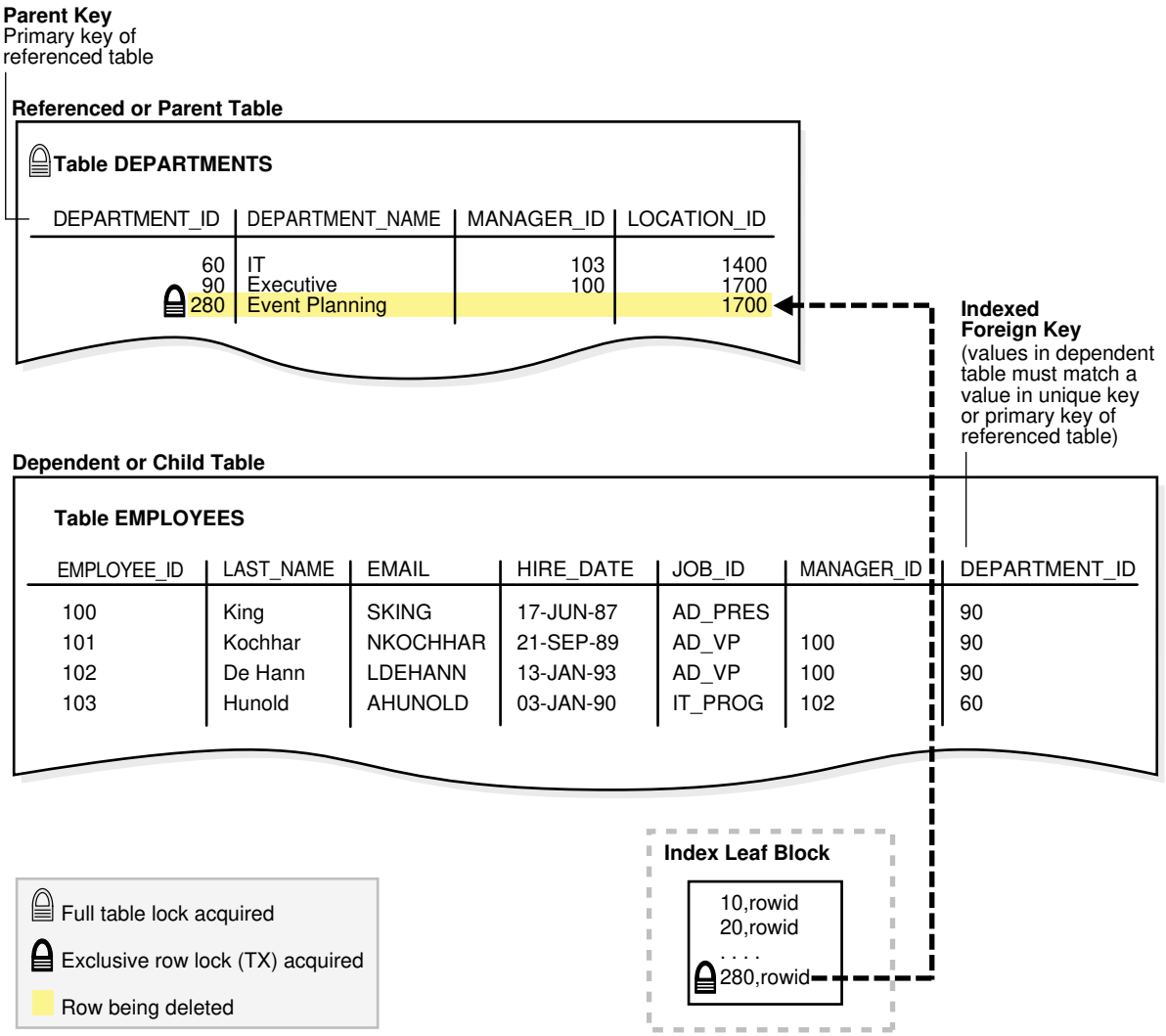
## Locks and Indexed Foreign Keys

The database does *not* acquire a full table lock on the child table when a foreign key column in the child table *is* indexed, and a session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

A lock on the parent table prevents transactions from acquiring exclusive table locks, but does not prevent DML on the parent *or* child table during the primary key modification. This situation is preferable if primary key modifications occur on the parent table while updates occur on the child table.

Figure 11-4 shows child table `employees` with an indexed `department_id` column. A transaction deletes department 280 from `departments`. This deletion does not cause the database to acquire a full table lock on the `employees` table as in the scenario described in "Locks and Unindexed Foreign Keys".

**Figure 11-4 Locking Mechanisms with Indexed Foreign Key**




If the child table specifies `ON DELETE CASCADE`, then deletions from the parent table can result in deletions from the child table. For example, the deletion of department 280 can cause the deletion of records from `employees` for employees in the deleted department. In this case, waiting and locking rules are the same as if you deleted rows from the child table after deleting rows from the parent table.

**See Also:**

- "Foreign Key Constraints"
- "Introduction to Indexes"

# DDL Locks

A **data dictionary (DDL) lock** protects the definition of a schema object while an ongoing DDL operation acts on or refers to the object.

Only individual schema objects that are modified or referenced are locked during DDL operations. The database never locks the whole data dictionary.

Oracle Database acquires a DDL lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. For example, if a user creates a stored procedure, then Oracle Database automatically acquires DDL locks for all schema objects referenced in the procedure definition. The DDL locks prevent these objects from being altered or dropped before procedure compilation is complete.

- Exclusive DDL Locks
  An exclusive DDL lock prevents other sessions from obtaining a DDL or DML lock.
- Share DDL Locks
  A share DDL lock for a resource prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.
- Breakable Parse Locks
  A **parse lock** is held by a SQL statement or PL/SQL program unit for each schema object that it references.

## Exclusive DDL Locks

An exclusive DDL lock prevents other sessions from obtaining a DDL or DML lock.

Most DDL operations require exclusive DDL locks for a resource to prevent destructive interference with other DDL operations that might modify or reference the same schema object. For example, `DROP TABLE` is not allowed to drop a table while `ALTER TABLE` is adding a column to it, and vice versa.

Exclusive DDL locks last for the duration of DDL statement execution and automatic commit. During the acquisition of an exclusive DDL lock, if another DDL lock is held on the schema object by another operation, then the acquisition waits until the older DDL lock is released and then proceeds.

> **See Also:**
>
> "Share DDL Locks" describes situations where exclusive locks are not required to prevent destructive interference

## Share DDL Locks

A share DDL lock for a resource prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.

For example, when a `CREATE PROCEDURE` statement is run, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table.

A share DDL lock lasts for the duration of DDL statement execution and automatic commit. Thus, a transaction holding a share DDL lock is guaranteed that the definition of the referenced schema object remains constant during the transaction.

### Breakable Parse Locks

A **parse lock** is held by a SQL statement or PL/SQL program unit for each schema object that it references.

Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped. A parse lock is called a *breakable parse lock* because it does not disallow any DDL operation and can be broken to allow conflicting DDL operations.

A parse lock is acquired in the shared pool during the parse phase of SQL statement execution. The lock is held as long as the shared SQL area for that statement remains in the shared pool.

> **See Also:**
>
> "Shared Pool"

## System Locks

Oracle Database uses various types of system locks to protect internal database and memory structures. These mechanisms are inaccessible to users because users have no control over their occurrence or duration.

- Latches
  A **latch** is a simple, low-level serialization mechanism that coordinates multiuser access to shared data structures, objects, and files.
- Mutexes
  A **mutual exclusion object (mutex)** is a low-level mechanism that prevents an object in memory from aging out or from being corrupted when accessed by concurrent processes. A mutex is similar to a latch, but whereas a latch typically protects a group of objects, a mutex protects a single object.
- Internal Locks
  Internal locks are higher-level, more complex mechanisms than latches and mutexes and serve various purposes.

### Latches

A **latch** is a simple, low-level serialization mechanism that coordinates multiuser access to shared data structures, objects, and files.

Latches protect shared memory resources from corruption when accessed by multiple processes. Specifically, latches protect data structures from the following situations:

- Concurrent modification by multiple sessions
- Being read by one session while being modified by another session
- Deallocation (aging out) of memory while being accessed

Typically, a single latch protects multiple objects in the SGA. For example, background processes such as DBW and LGWR allocate memory from the shared pool to create data structures. To allocate this memory, these processes use a shared pool latch that serializes access to prevent two processes from trying to inspect or modify the shared pool simultaneously. After the memory is allocated, other processes may need to access shared pool areas such as the library cache, which is required for parsing. In this case, processes latch only the library cache, not the entire shared pool.

Unlike enqueue latches such as row locks, latches do not permit sessions to queue. When a latch becomes available, the first session to request the latch obtains exclusive access to it. The phenomenon of latch spinning occurs when a process repeatedly requests a latch in a loop, whereas latch sleeping occurs when a process releases the CPU before renewing the latch request.

Typically, an Oracle process acquires a latch for an extremely short time while manipulating or looking at a data structure. For example, while processing a salary update of a single employee, the database may obtain and release thousands of latches. The implementation of latches is operating system-dependent, especially in respect to whether and how long a process waits for a latch.

An increase in latching means a decrease in concurrency. For example, excessive hard parse operations create contention for the library cache latch. The `V$LATCH` view contains detailed latch usage statistics for each latch, including the number of times each latch was requested and waited for.

**See Also:**

- "SQL Parsing"
- *Oracle Database Reference* to learn about `V$LATCH`
- *Oracle Database Performance Tuning Guide* to learn about wait event statistics

## Mutexes

A **mutual exclusion object (mutex)** is a low-level mechanism that prevents an object in memory from aging out or from being corrupted when accessed by concurrent processes. A mutex is similar to a latch, but whereas a latch typically protects a group of objects, a mutex protects a single object.

Mutexes provide several benefits:

- A mutex can reduce the possibility of contention.

  Because a latch protects multiple objects, it can become a bottleneck when processes attempt to access any of these objects concurrently. By serializing access to an individual object rather than a group, a mutex increases availability.
- A mutex consumes less memory than a latch.
- When in shared mode, a mutex permits concurrent reference by multiple sessions.

## Internal Locks

Internal locks are higher-level, more complex mechanisms than latches and mutexes and serve various purposes.

The database uses the following types of internal locks:

- Dictionary cache locks

  These locks are of very short duration and are held on entries in dictionary caches while the entries are being modified or used. They guarantee that statements being parsed do not see inconsistent object definitions. Dictionary cache locks can be shared or exclusive. Shared locks are released when the parse is complete, whereas exclusive locks are released when the DDL operation is complete.

- File and log management locks

  These locks protect various files. For example, an internal lock protects the control file so that only one process at a time can change it. Another lock coordinates the use and archiving of the online redo log files. Data files are locked to ensure that multiple instances mount a database in shared mode or that one instance mounts it in exclusive mode. Because file and log locks indicate the status of files, these locks are necessarily held for a long time.

- Tablespace and undo segment locks

  These locks protect tablespaces and undo segments. For example, all instances accessing a database must agree on whether a tablespace is online or offline. Undo segments are locked so that only one database instance can write to a segment.

> **See Also:**
>
> "Data Dictionary Cache"

# Overview of Manual Data Locks

You can manually override the Oracle Database default locking mechanisms.

Oracle Database performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, overriding the default locking is useful in situations such as the following:

- Applications require transaction-level read consistency or repeatable reads.

  In this case, queries must produce consistent data for the duration of the transaction, not reflecting changes by other transactions. You can achieve transaction-level read consistency by using explicit locking, read-only transactions, serializable transactions, or by overriding default locking.

- Applications require that a transaction have exclusive access to a resource so that the transaction does not have to wait for other transactions to complete.

You can override Oracle Database automatic locking at the session or transaction level. At the session level, a session can set the required transaction isolation level with the `ALTER SESSION` statement. At the transaction level, transactions that include the following SQL statements override Oracle Database default locking:

- The `SET TRANSACTION ISOLATION LEVEL` statement
- The `LOCK TABLE` statement (which locks either a table or, when used with views, the base tables)
- The `SELECT ... FOR UPDATE` statement

Locks acquired by the preceding statements are released after the transaction ends or a rollback to savepoint releases them.

If Oracle Database default locking is overridden at any level, then the database administrator or application developer should ensure that the overriding locking procedures operate correctly. The locking procedures must satisfy the following criteria: data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or are appropriately handled.

**See Also:**

- *Oracle Database SQL Language Reference* for descriptions of `LOCK TABLE` and `SELECT`
- *Oracle Database Development Guide* to learn how to manually lock tables

# Overview of User-Defined Locks

With Oracle Database Lock Management services, you can define your own locks for a specific application.

For example, you might create a lock to serialize access to a message log on the file system. Because a reserved user lock is the same as an Oracle Database lock, it has all the Oracle Database lock functionality including deadlock detection. User locks never conflict with Oracle Database locks, because they are identified with the prefix `UL`.

The Oracle Database Lock Management services are available through procedures in the `DBMS_LOCK` package. You can include statements in PL/SQL blocks that:

- Request a lock of a specific type
- Give the lock a unique name recognizable in another procedure in the same or in another instance
- Change the lock type
- Release the lock

**See Also:**

- *Oracle Database Development Guide* for more information about Oracle Database Lock Management services
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_LOCK`

12

# Transactions

This chapter defines a transaction and describes how the database processes transactions.

This chapter contains the following sections:


- Introduction to Transactions
- Overview of Transaction Control
- Overview of Transaction Guard
- Overview of Application Continuity
- Overview of Autonomous Transactions
- Overview of Distributed Transactions


- Introduction to Transactions
  A **transaction** is a logical, atomic unit of work that contains one or more SQL statements.
- Overview of Transaction Control
  **Transaction control** is the management of changes made by DML statements and the grouping of DML statements into transactions.
- Overview of Transaction Guard
  **Transaction Guard** is an API that applications can use to provide **transaction idempotence**, which is the ability of the database to preserve a guaranteed commit outcome that indicates whether a transaction committed and completed. Oracle Database provides the API for JDBC thin, OCI, OCCI, and ODP.Net.
- Overview of Application Continuity
  **Application Continuity** attempts to mask outages from applications by replaying incomplete application requests after unplanned and planned outages. In this context, a request is a unit of work from the application.
- Overview of Autonomous Transactions
  An **autonomous transaction** is an independent transaction that can be called from another transaction, which is the main transaction. You can suspend the calling transaction, perform SQL operations and commit or undo them in the autonomous transaction, and then resume the calling transaction.
- Overview of Distributed Transactions
  A **distributed transaction** is a transaction that includes one or more statements that update data on two or more distinct nodes of a distributed database, using a schema object called a database link.

## Introduction to Transactions

A **transaction** is a logical, atomic unit of work that contains one or more SQL statements.

A transaction groups SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone from the database. Oracle Database assigns every transaction a unique identifier called a transaction ID.

All Oracle transactions obey the basic properties of a database transaction, known as ACID properties. ACID is an acronym for the following:

- Atomicity

  All tasks of a transaction are performed or none of them are. There are no partial transactions. For example, if a transaction starts updating 100 rows, but the system fails after 20 updates, then the database rolls back the changes to these 20 rows.

- Consistency

  The transaction takes the database from one consistent state to another consistent state. For example, in a banking transaction that debits a savings account and credits a checking account, a failure must not cause the database to credit only one account, which would lead to inconsistent data.

- Isolation

  The effect of a transaction is not visible to other transactions until the transaction is committed. For example, one user updating the `hr.employees` table does not see the uncommitted changes to `employees` made concurrently by another user. Thus, it appears to users as if transactions are executing serially.

- Durability

  Changes made by committed transactions are permanent. After a transaction completes, the database ensures through its recovery mechanisms that changes from the transaction are not lost.

The use of transactions is one of the most important ways that a database management system differs from a file system.

- Sample Transaction: Account Debit and Credit
  To illustrate the concept of a transaction, consider a banking database.
- Structure of a Transaction
  A database transaction consists of one or more statements.
- Statement-Level Atomicity
  Oracle Database supports **statement-level atomicity**, which means that a SQL statement is an atomic unit of work and either completely succeeds or completely fails.
- System Change Numbers (SCNs)
  A **system change number (SCN)** is a logical, internal time stamp used by Oracle Database.

## Sample Transaction: Account Debit and Credit

To illustrate the concept of a transaction, consider a banking database.

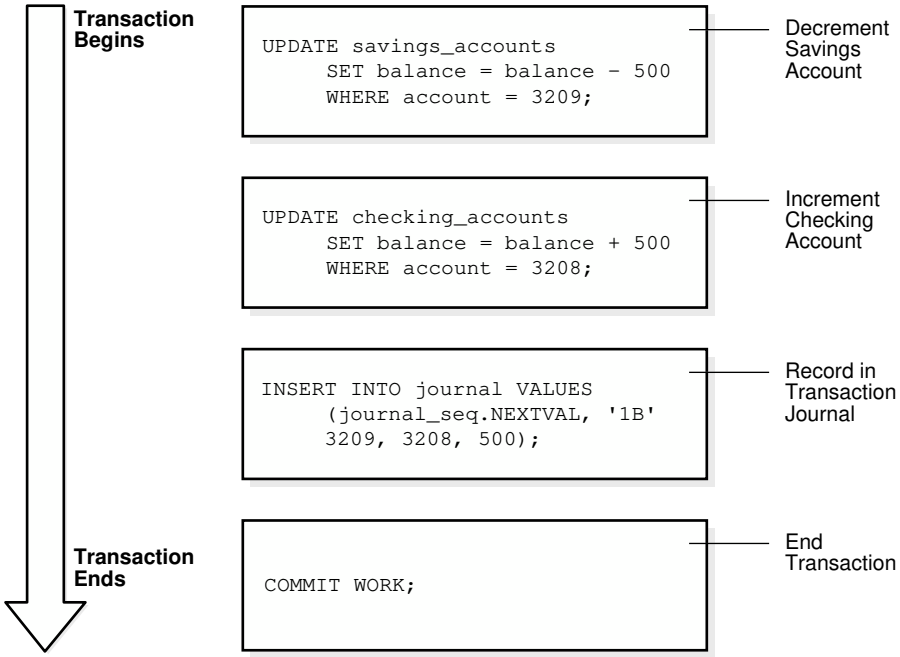
When a customer transfers money from a savings account to a checking account, the transaction must consist of three separate operations:

- Decrement the savings account
- Increment the checking account
- Record the transaction in the transaction journal

Oracle Database must allow for two situations. If all three SQL statements maintain the accounts in proper balance, then the effects of the transaction can be applied to the database. However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, then the database must roll back the entire transaction so that the balance of all accounts is correct.

The following graphic illustrates a banking transaction. The first statement subtracts $500 from savings account 3209. The second statement adds $500 to checking account 3208. The third statement inserts a record of the transfer into the journal table. The final statement commits the transaction.

**Figure 12-1 A Banking Transaction**




## Structure of a Transaction

A database transaction consists of one or more statements.

Specifically, a transaction consists of one of the following:

- One or more data manipulation language (DML) statements that together constitute an atomic change to the database
- One data definition language (DDL) statement

A transaction has a beginning and an end.

- Beginning of a Transaction
  A transaction begins when the first executable SQL statement is encountered.
- End of a Transaction
  A transaction can end under different circumstances.

> **See Also:**
>
> - "Overview of SQL Statements"
> - *Oracle Database SQL Language Reference* for an account of the types of SQL statements

## Beginning of a Transaction

A transaction begins when the first executable SQL statement is encountered.

An executable SQL statement is a SQL statement that generates calls to a database instance, including DML and DDL statements and the `SET TRANSACTION` statement.

When a transaction begins, Oracle Database assigns the transaction to an available undo data segment to record the undo entries for the new transaction. A transaction ID is not allocated until an undo segment and transaction table slot are allocated, which occurs during the first DML statement. A transaction ID is unique to a transaction and represents the undo segment number, slot, and sequence number.

The following example execute an `UPDATE` statement to begin a transaction and queries `V$TRANSACTION` for details about the transaction:

```
SQL> UPDATE hr.employees SET salary=salary;

107 rows updated.

SQL> SELECT XID AS "txn id", XIDUSN AS "undo seg", XIDSLOT AS "slot",
  2  XIDSQN AS "seq", STATUS AS "txn status"
  3  FROM V$TRANSACTION;

txn id             undo seg       slot        seq txn status
---------------- ---------- ---------- ---------- ----------------
0600060037000000          6          6         55 ACTIVE
```

> **See Also:**
>
> "Undo Segments"

## End of a Transaction

A transaction can end under different circumstances.

A transaction ends when any of the following actions occurs:

- A user issues a `COMMIT` or `ROLLBACK` statement *without* a `SAVEPOINT` clause.

  In a commit, a user explicitly or implicitly requested that the changes in the transaction be made permanent. Changes made by the transaction are permanent and visible to other users only after a transaction commits.

- A user runs a DDL command such as `CREATE`, `DROP`, `RENAME`, or `ALTER`.

  The database issues an implicit `COMMIT` statement before and after every DDL statement. If the current transaction contains DML statements, then Oracle Database first commits the transaction and then runs and commits the DDL statement as a new, single-statement transaction.

- A user exits normally from most Oracle Database utilities and tools, causing the current transaction to be implicitly committed. The commit behavior when a user disconnects is application-dependent and configurable.

> **Note:**
>
> Applications should always explicitly commit or undo transactions before program termination.

- A client process terminates abnormally, causing the transaction to be implicitly rolled back using metadata stored in the transaction table and the undo segment.

After one transaction ends, the next executable SQL statement automatically starts the following transaction. The following example executes an `UPDATE` to start a transaction, ends the transaction with a `ROLLBACK` statement, and then executes an `UPDATE` to start a new transaction (note that the transaction IDs are different):

```
SQL> UPDATE hr.employees SET salary=salary;
107 rows updated.

SQL> SELECT XID, STATUS FROM V$TRANSACTION;

XID              STATUS
---------------- ----------------
0800090033000000 ACTIVE

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT XID FROM V$TRANSACTION;

no rows selected

SQL> UPDATE hr.employees SET last_name=last_name;

107 rows updated.

SQL> SELECT XID, STATUS FROM V$TRANSACTION;

XID              STATUS
---------------- ----------------
0900050033000000 ACTIVE
```

> **See Also:**
>
> - "Sample Transaction: Account Debit and Credit" for an example of a transaction that ends with a commit.
> - *Oracle Database SQL Language Reference* to learn about `COMMIT`

## Statement-Level Atomicity

Oracle Database supports **statement-level atomicity**, which means that a SQL statement is an atomic unit of work and either completely succeeds or completely fails.

A successful statement is different from a committed transaction. A single SQL statement executes successfully if the database parses and runs it without error as an atomic unit, as when all rows are changed in a multirow update.

If a SQL statement causes an error during execution, then it is not successful and so all effects of the statement are rolled back. This operation is a statement-level rollback. This operation has the following characteristics:

- A SQL statement that does not succeed causes the loss only of work it would have performed itself.

  The unsuccessful statement does not cause the loss of any work that preceded it in the current transaction. For example, if the execution of the second `UPDATE` statement in "Sample Transaction: Account Debit and Credit" causes an error and is rolled back, then the work performed by the first `UPDATE` statement is not rolled back. The first `UPDATE` statement can be committed or rolled back explicitly by the user.
- The effect of the rollback is as if the statement had never been run.

  Any side effects of an atomic statement, for example, triggers invoked upon execution of the statement, are considered part of the atomic statement. Either all work generated as part of the atomic statement succeeds or none does.

An example of an error causing a statement-level rollback is an attempt to insert a duplicate primary key. Single SQL statements involved in a deadlock, which is competition for the same data, can also cause a statement-level rollback. However, errors discovered during SQL statement parsing, such as a syntax error, have not yet been run and so do not cause a statement-level rollback.

> **See Also:**
>
> - "SQL Parsing"
> - "Locks and Deadlocks"
> - "Overview of Triggers"

## System Change Numbers (SCNs)

A **system change number (SCN)** is a logical, internal time stamp used by Oracle Database.

SCNs order events that occur within the database, which is necessary to satisfy the ACID properties of a transaction. Oracle Database uses SCNs to mark the SCN before which all changes are known to be on disk so that recovery avoids applying unnecessary redo. The database also uses SCNs to mark the point at which no redo exists for a set of data so that recovery can stop.

SCNs occur in a monotonically increasing sequence. Oracle Database can use an SCN like a clock because an observed SCN indicates a logical point in time, and repeated observations return equal or greater values. If one event has a lower SCN than another event, then it

occurred at an earlier time in the database. Several events may share the same SCN, which means that they occurred at the same time in the database.

Every transaction has an SCN. For example, if a transaction updates a row, then the database records the SCN at which this update occurred. Other modifications in this transaction have the same SCN. When a transaction commits, the database records an SCN for this commit.

Oracle Database increments SCNs in the system global area (SGA). When a transaction modifies data, the database writes a new SCN to the undo data segment assigned to the transaction. The log writer process then writes the commit record of the transaction immediately to the online redo log. The commit record has the unique SCN of the transaction. Oracle Database also uses SCNs as part of its instance recovery and media recovery mechanisms.

> **See Also:**
>
> "Overview of Instance Recovery"

# Overview of Transaction Control

**Transaction control** is the management of changes made by DML statements and the grouping of DML statements into transactions.

In general, application designers are concerned with transaction control so that work is accomplished in logical units and data is kept consistent.

Transaction control involves using the following statements, as described in "Transaction Control Statements":

- The `COMMIT` statement ends the current transaction and makes all changes performed in the transaction permanent. `COMMIT` also erases all savepoints in the transaction and releases transaction locks.
- The `ROLLBACK` statement reverses the work done in the current transaction; it causes all data changes since the last `COMMIT` or `ROLLBACK` to be discarded. The `ROLLBACK TO SAVEPOINT` statement undoes the changes since the last savepoint but does not end the entire transaction.
- The `SAVEPOINT` statement identifies a point in a transaction to which you can later roll back.

The session in Table 12-1 illustrates the basic concepts of transaction control.

**Table 12-1 Transaction Control**

| T | Session | Explanation |
|---|---|---|
| t0 | `COMMIT;` | This statement ends any existing transaction in the session. |
| t1 | `SET TRANSACTION NAME 'sal_update';` | This statement begins a transaction and names it `sal_update`. |

**Table 12-1 (Cont.) Transaction Control**

| T | Session | Explanation |
|---|---|---|
| t2 | `UPDATE employees`<br>`SET salary = 7000`<br>`WHERE last_name = 'Banda';` | This statement updates the salary for Banda to 7000. |
| t3 | `SAVEPOINT after_banda_sal;` | This statement creates a savepoint named `after_banda_sal`, enabling changes in this transaction to be rolled back to this point. |
| t4 | `UPDATE employees`<br>`SET salary = 12000`<br>`WHERE last_name = 'Greene';` | This statement updates the salary for Greene to 12000. |
| t5 | `SAVEPOINT after_greene_sal;` | This statement creates a savepoint named `after_greene_sal`, enabling changes in this transaction to be rolled back to this point. |
| t6 | `ROLLBACK TO SAVEPOINT`<br>`after_banda_sal;` | This statement rolls back the transaction to t3, undoing the update to Greene's salary at t4. The `sal_update` transaction has *not* ended. |
| t7 | `UPDATE employees`<br>`SET salary = 11000`<br>`WHERE last_name = 'Greene';` | This statement updates the salary for Greene to 11000 in transaction `sal_update`. |
| t8 | `ROLLBACK;` | This statement rolls back all changes in transaction `sal_update`, ending the transaction. |
| t9 | `SET TRANSACTION NAME 'sal_update2';` | This statement begins a new transaction in the session and names it `sal_update2`. |
| t10 | `UPDATE employees`<br>`SET salary = 7050`<br>`WHERE last_name = 'Banda';` | This statement updates the salary for Banda to 7050. |
| t11 | `UPDATE employees`<br>`SET salary = 10950`<br>`WHERE last_name = 'Greene';` | This statement updates the salary for Greene to 10950. |

**Table 12-1 (Cont.) Transaction Control**

| T | Session | Explanation |
|---|---|---|
| t12 | `COMMIT;` | This statement commits all changes made in transaction `sal_update2`, ending the transaction. The commit guarantees that the changes are saved in the online redo log files. |

- Transaction Names
  A transaction name is an optional, user-specified tag that serves as a reminder of the work that the transaction is performing. You name a transaction with the `SET TRANSACTION ... NAME` statement, which if used must be first statement of the transaction.
- Active Transactions
  An **active transaction** is one that has started but not yet committed or rolled back.
- Savepoints
  A **savepoint** is a user-declared intermediate marker within the context of a transaction.
- Rollback of Transactions
  A rollback of an uncommitted transaction undoes any changes to data that have been performed by SQL statements within the transaction.
- Commits of Transactions
  A commit ends the current transaction and makes permanent all changes performed in the transaction.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about transaction control statements

## Transaction Names

A transaction name is an optional, user-specified tag that serves as a reminder of the work that the transaction is performing. You name a transaction with the `SET TRANSACTION ... NAME` statement, which if used must be first statement of the transaction.

In Table 12-1, the first transaction was named `sal_update` and the second was named `sal_update2`.

Transaction names provide the following advantages:

- It is easier to monitor long-running transactions and to resolve in-doubt distributed transactions.
- You can view transaction names along with transaction IDs in applications. For example, a database administrator can view transaction names in Oracle Enterprise Manager (Enterprise Manager) when monitoring system activity.
- The database writes transaction names to the transaction auditing redo record, so you can use LogMiner to search for a specific transaction in the redo log.

- You can use transaction names to find a specific transaction in data dictionary views such as `V$TRANSACTION`.

> **See Also:**
>
> - *Oracle Database Reference* to learn about `V$TRANSACTION`
> - *Oracle Database SQL Language Reference* to learn about `SET TRANSACTION`

## Active Transactions

An **active transaction** is one that has started but not yet committed or rolled back.

In Table 12-1, the first statement to modify data in the `sal_update` transaction is the update to Banda's salary. From the successful execution of this update until the `ROLLBACK` statement ends the transaction, the `sal_update` transaction is active.

Data changes made by a transaction are temporary until the transaction is committed or rolled back. Before the transaction ends, the state of the data is as shown in the following table.

**Table 12-2 State of the Data Before the Transaction Ends**

| State | Description | To Learn More |
|---|---|---|
| Oracle Database has generated undo information in the SGA. | The undo data contains the old data values changed by the SQL statements of the transaction. | "Read Consistency in the Read Committed Isolation Level" |
| Oracle Database has generated redo in the online redo log buffer of the SGA. | The redo log record contains the change to the data block and the change to the undo block. | "Redo Log Buffer " |
| Changes have been made to the database buffers of the SGA. | The data changes for a committed transaction, stored in the database buffers of the SGA, are not necessarily written immediately to the data files by the database writer (DBW). The disk write can happen before or after the commit. | "Database Buffer Cache" |
| The rows affected by the data change are locked. | Other users cannot change the data in the affected rows, nor can they see the uncommitted changes. | "Summary of Locking Behavior" |

## Savepoints

A **savepoint** is a user-declared intermediate marker within the context of a transaction.

Internally, the savepoint marker resolves to an SCN. Savepoints divide a long transaction into smaller parts.

If you use savepoints in a long transaction, then you have the option later of rolling back work performed before the current point in the transaction but after a declared savepoint within the transaction. Thus, if you make an error, you do not need to resubmit every statement.

Table 12-1 creates savepoint `after_banda_sal` so that the update to the Greene salary can be rolled back to this savepoint.

- Rollback to Savepoint
  A rollback to a savepoint in an uncommitted transaction means undoing any changes made after the specified savepoint, but it does not mean a rollback of the transaction itself.
- Enqueued Transactions
  Depending on the scenario, transactions waiting for previously locked resources may still be blocked after a rollback to savepoint.

## Rollback to Savepoint

A rollback to a savepoint in an uncommitted transaction means undoing any changes made after the specified savepoint, but it does not mean a rollback of the transaction itself.

When a transaction is rolled back to a savepoint, as when the `ROLLBACK TO SAVEPOINT after_banda_sal` is run in Table 12-1, the following occurs:

1. Oracle Database rolls back only the statements run after the savepoint.

   In Table 12-1, the `ROLLBACK TO SAVEPOINT` causes the `UPDATE` for Greene to be rolled back, but not the `UPDATE` for Banda.

2. Oracle Database preserves the savepoint specified in the `ROLLBACK TO SAVEPOINT` statement, but all subsequent savepoints are lost.

   In Table 12-1, the `ROLLBACK TO SAVEPOINT` causes the `after_greene_sal` savepoint to be lost.

3. Oracle Database releases all table and row locks acquired after the specified savepoint but retains all data locks acquired before the savepoint.

The transaction remains active and can be continued.

**See Also:**

- *Oracle Database SQL Language Reference* to learn about the `ROLLBACK` and `SAVEPOINT` statements
- *Oracle Database PL/SQL Language Reference* to learn about transaction processing and control

## Enqueued Transactions

Depending on the scenario, transactions waiting for previously locked resources may still be blocked after a rollback to savepoint.

When a transaction is blocked by another transaction it enqueues on the blocking transaction itself, so that the entire blocking transaction must commit or roll back for the blocked transaction to continue.

In the scenario shown in the following table, session 1 rolls back to a savepoint created before it executed a DML statement. However, session 2 is still blocked because it is waiting for the session 1 transaction to complete.

**Table 12-3 Rollback to Savepoint Example**

| T | Session 1 | Session 2 | Session 3 | Explanation |
|---|---|---|---|---|
| t0 | `UPDATE employees SET salary=7000 WHERE last_name= 'Banda';` | | | Session 1 begins a transaction. The session places an exclusive lock on the `Banda` row (TX) and a subexclusive table lock (SX) on the table. |
| t1 | `SAVEPOINT after_banda_sal;` | | | Session 1 creates a savepoint named `after_banda_sal`. |
| t2 | `UPDATE employees SET salary=12000 WHERE last_name= 'Greene';` | | | Session 1 locks the `Greene` row. |
| t3 | | `UPDATE employees SET salary=14000 WHERE last_name= 'Greene';` | | Session 2 attempts to update the `Greene` row, but fails to acquire a lock because session 1 has a lock on this row. No transaction has begun in session 2. |
| t4 | `ROLLBACK TO SAVEPOINT after_banda_sal;` | | | Session 1 rolls back the update to the salary for `Greene`, which releases the row lock for `Greene`. The table lock acquired at t0 is not released.<br>At this point, session 2 is *still* blocked by session 1 because session 2 enqueues on the session 1 *transaction*, which has not yet completed. |
| t5 | | | `UPDATE employees SET salary=11000 WHERE last_name= 'Greene';` | The `Greene` row is currently unlocked, so session 3 acquires a lock for an update to the `Greene` row. This statement begins a transaction in session 3. |
| t6 | `COMMIT;` | | | Session 1 commits, ending its transaction. Session 2 is now enqueued for its update to the `Greene` row behind the transaction in session 3. |

> **See Also:**
>
> "Lock Duration" to learn more about when Oracle Database releases locks

## Rollback of Transactions

A rollback of an uncommitted transaction undoes any changes to data that have been performed by SQL statements within the transaction.

After a transaction has been rolled back, the effects of the work done in the transaction no longer exist. In rolling back an entire transaction, without referencing any savepoints, Oracle Database performs the following actions:

- Undoes all changes made by all the SQL statements in the transaction by using the corresponding undo segments

  The transaction table entry for every active transaction contains a pointer to all the undo data (in reverse order of application) for the transaction. The database reads the data from the undo segment, reverses the operation, and then marks the undo entry as applied. Thus, if a transaction inserts a row, then a rollback deletes it. If a transaction updates a row, then a rollback reverses the update. If a transaction deletes a row, then a rollback reinserts it. In Table 12-1, the `ROLLBACK` reverses the updates to the salaries of Greene and Banda.
- Releases all the locks of data held by the transaction
- Erases all savepoints in the transaction

  In Table 12-1, the `ROLLBACK` deletes the savepoint `after_banda_sal`. The `after_greene_sal` savepoint was removed by the `ROLLBACK TO SAVEPOINT` statement.
- Ends the transaction

  In Table 12-1, the `ROLLBACK` leaves the database in the same state as it was after the initial `COMMIT` was executed.

The duration of a rollback is a function of the amount of data modified.

> **See Also:**
>
> - "Undo Segments"
> - *Oracle Database SQL Language Reference* to learn about the `ROLLBACK` command

## Commits of Transactions

A commit ends the current transaction and makes permanent all changes performed in the transaction.

In Table 12-1, a second transaction begins with `sal_update2` and ends with an explicit `COMMIT` statement. The changes that resulted from the two `UPDATE` statements are now made permanent.

When a transaction commits, the following actions occur:

- The database generates an SCN for the `COMMIT`.

  The internal transaction table for the associated undo tablespace records that the transaction has committed. The corresponding unique SCN of the transaction is assigned and recorded in the transaction table.

- The log writer process (LGWR) process writes remaining redo log entries in the redo log buffers to the online redo log and writes the transaction SCN to the online redo log. *This atomic event constitutes the commit of the transaction.*

- Oracle Database releases locks held on rows and tables.

  Users who were enqueued waiting on locks held by the uncommitted transaction are allowed to proceed with their work.

- Oracle Database deletes savepoints.

  In Table 12-1, no savepoints existed in the `sal_update` transaction so no savepoints were erased.

- Oracle Database performs a commit cleanout.

  If modified blocks containing data from the committed transaction are still in the SGA, and if no other session is modifying them, then the database removes lock-related transaction information (the ITL entry) from the blocks.

  Ideally, the `COMMIT` cleans the blocks so that a subsequent `SELECT` does not have to perform this task. If no ITL entry exists for a specific row, then it is not locked. If an ITL entry exists for a specific row, then it is possibly locked, so a session must check the undo segment header to determine whether this interested transaction has committed. If the interested transaction has committed, then the session cleans out the block, which generates redo. However, if the `COMMIT` cleaned out the ITL previously, then the check and cleanout are unnecessary.

  > **Note:**
  >
  > Because a block cleanout generates redo, a query may generate redo and thus cause blocks to be written during the next checkpoint.

- Oracle Database marks the transaction complete.

After a transaction commits, users can view the changes.

Typically, a commit is a fast operation, regardless of the transaction size. The speed of a commit does not increase with the size of the data modified in the transaction. The lengthiest part of the commit is the physical disk I/O performed by LGWR. However, the amount of time spent by LGWR is reduced because it has been incrementally writing the contents of the redo log buffer in the background.

The default behavior is for LGWR to write redo to the online redo log synchronously and for transactions to wait for the buffered redo to be on disk before returning a commit to the user. However, for lower transaction commit latency, application developers can specify that redo be written asynchronously so that transactions need not wait for the redo to be on disk and can return from the `COMMIT` call immediately.

**See Also:**

- "Serializable Isolation Level"
- "Locking Mechanisms"
- "Overview of Background Processes" for more information about LGWR
- *Oracle Database PL/SQL Packages and Types Reference* for more information on asynchronous commit

# Overview of Transaction Guard

**Transaction Guard** is an API that applications can use to provide **transaction idempotence**, which is the ability of the database to preserve a guaranteed commit outcome that indicates whether a transaction committed and completed. Oracle Database provides the API for JDBC thin, OCI, OCCI, and ODP.Net.

A recoverable error is caused by an external system failure, independent of the application session logic that is executing. Recoverable errors occur following planned and unplanned outages of foreground processes, networks, nodes, storage, and databases. If an outage breaks the connection between a client application and the database, then the application receives a disconnection error message. The transaction that was running when the connection broke is called an in-flight transaction.

To decide whether to resubmit the transaction or to return the result (committed or uncommitted) to the client, the application must determine the outcome of the in-flight transaction. Before Oracle Database 12c, commit messages returned to the client were not persistent. Checking a transaction was no guarantee that it would not commit after being checked, permitting duplicate transactions and other forms of logical corruption. For example, a user might refresh a web browser when purchasing a book online and be charged twice for the same book.

- Benefits of Transaction Guard
  Starting in Oracle Database 12c, Transaction Guard provides applications with a tool for determining the status of an in-flight transaction following a recoverable outage.
- How Transaction Guard Works
  This section explains the problem of lost commit messages and how Transaction Guard uses logical transaction IDs to solve the problem.
- Transaction Guard: Example
  In this scenario, the commit message is lost because of a recoverable error.

**See Also:**

- "Introduction to Transactions "
- *Oracle Database Development Guide* to learn about Transaction Guard
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn how to configure services for Transaction Guard

# Benefits of Transaction Guard

Starting in Oracle Database 12c, Transaction Guard provides applications with a tool for determining the status of an in-flight transaction following a recoverable outage.

Using Transaction Guard, an application can ensure that a transaction executes no more than once. For example, if an online bookstore application determines that the previously submitted commit failed, then the application can safely resubmit.

Transaction Guard provides a tool for at-most-once execution to avoid the application executing duplicate submissions. Transaction Guard provides a known outcome for every transaction.

Transaction Guard is a core Oracle Database capability. Application Continuity uses Transaction Guard when masking outages from end users. Without Transaction Guard, an application retrying after an error may cause duplicate transactions to be committed.

**See Also:**

- "Overview of Application Continuity" to learn about Application Continuity, which works with Transaction Guard to help developers achieve high application availability
- *Oracle Database Development Guide* to learn about Transaction Guard, including the types of supported and included transactions

# How Transaction Guard Works

This section explains the problem of lost commit messages and how Transaction Guard uses logical transaction IDs to solve the problem.

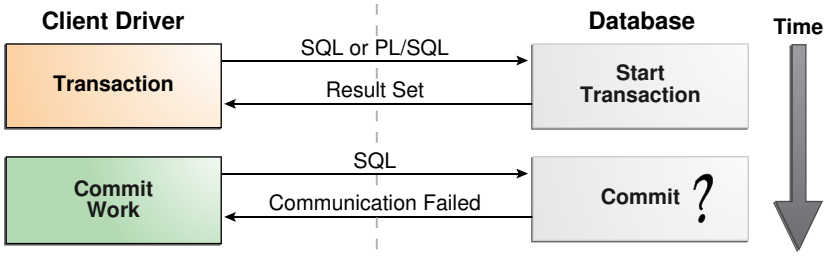
- Lost Commit Messages
  When designing for idempotence, developers must address the problem of communication failures after submission of commit statements. Commit messages do not persist in the database and so cannot be retrieved after a failure.
- Logical Transaction ID
  Oracle Database solves the communication failure by using a globally unique identifier called a **logical transaction ID**.

## Lost Commit Messages

When designing for idempotence, developers must address the problem of communication failures after submission of commit statements. Commit messages do not persist in the database and so cannot be retrieved after a failure.

The following graphic is a high-level representation of an interaction between a client application and a database.

**Figure 12-2 Lost Commit Message**




In the standard commit case, the database commits a transaction and returns a success message to the client. In Figure 12-2, the client submits a commit statement and receives a message stating that communication failed. This type of failure can occur for several reasons, including a database instance failure or network outage. In this scenario, the client does not know the state of the transaction.

Following a communication failure, the database may still be running the submission and be unaware that the client disconnected. Checking the transaction state does not guarantee that an active transaction will not commit after being checked. If the client resends the commit because of this out-of-date information, then the database may repeat the transaction, resulting in logical corruption.

## Logical Transaction ID

Oracle Database solves the communication failure by using a globally unique identifier called a **logical transaction ID**.

This ID contains the logical session number allocated when a session first connects, and a running commit number that is updated each time the session commits or rolls back.[1] From the application perspective, the logical transaction ID uniquely identifies the last database transaction submitted on the session that failed.

For each round trip from the client in which one or more transactions are committed, the database stores a logical transaction ID. This ID can provide transaction idempotence for interactions between the application and the database for each round trip that commits data.

The at-most-once protocol enables access to the commit outcome by requiring the database to do the following:

- Maintain the logical transaction ID for the retention period agreed for retry
- Persist the logical transaction ID on commit

While a transaction is running, both the database and client hold the logical transaction ID. The database gives the client a logical transaction ID at authentication, when borrowing from a connection pool, and at each round trip from the client driver that executes one or more commit operations.

Before the application can determine the outcome of the last transaction following a recoverable error, the application obtains the logical transaction ID held at the client using Java, OCI, OCCI, or ODP.Net APIs. The application then invokes the PL/SQL procedure `DBMS_APP_CONT.GET_LTXID_OUTCOME` with the logical transaction ID to determine the outcome of the last submission: committed (`true` or `false`) and user call completed (`true` or `false`).

[1] For Oracle Real Application Clusters (Oracle RAC), the logical transaction ID includes the database instance number as a prefix.

When using Transaction Guard, the application can replay transactions when the error is recoverable and the last transaction on the session has not committed. The application can continue when the last transaction has committed and the user call has completed. The application can use Transaction Guard to return the known outcome to the client so that the client can decide the next action to take.

**See Also:**

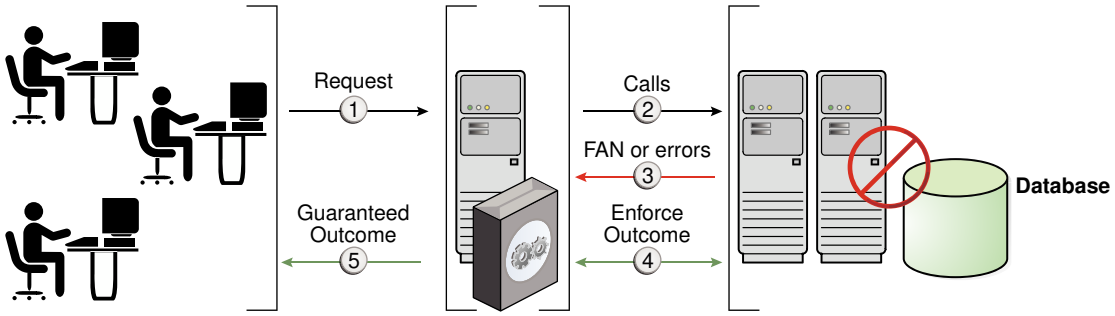
- *Oracle Database Development Guide* to learn about logical transaction IDs
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure

## Transaction Guard: Example

In this scenario, the commit message is lost because of a recoverable error.

Transaction Guard uses the logical transaction ID to preserve the outcome of the `COMMIT` statement, ensuring that there is a known outcome for the transaction.

**Figure 12-3 Check of Logical Transaction Status**




In Figure 12-3, the database informs the application whether the transaction committed and whether the last user call completed. The application can then return the result to the end user. The possibilities are:

- If the transaction committed and the user call completed, then the application can return the result to the end user and continue.
- If the transaction committed but the user call did not complete, then the application can return the result to the end user with warnings. Examples include a lost out bind or lost number of rows processed. Some applications depend on the extra information, whereas others do not.
- If the user call was not committed, then the application can return this information to the end user, or safely resubmit. The protocol is guaranteed. When the commit status returns false, the last submission is blocked from committing.

> **See Also:**
>
> *Oracle Database Development Guide* to learn how to use Transaction Guard

# Overview of Application Continuity

**Application Continuity** attempts to mask outages from applications by replaying incomplete application requests after unplanned and planned outages. In this context, a request is a unit of work from the application.

Typically, a request corresponds to the DML statements and other database calls of a single web request on a single database connection. In general, a request is demarcated by the calls made between check-out and check-in of a database connection from a connection pool.

This section contains the following topics:

- Benefits of Application Continuity
- Application Continuity Architecture

- Benefits of Application Continuity
  A basic problem for developers is how to mask a lost database session from end users.
- Application Continuity Architecture
  The key components of Application Continuity are runtime, reconnection, and replay.

## Benefits of Application Continuity

A basic problem for developers is how to mask a lost database session from end users.

Application Continuity attempts to solve the lost session problem by restoring the database session when any component disrupts the conversation between database and client. The restored database session includes all states, cursors, variables, and the most recent transaction when one exists.

- Use Case for Application Continuity
  In a typical case, a client has submitted a request to the database, which has built up both transactional and nontransactional states.
- Application Continuity for Planned Maintenance
  Application Continuity for planned outages enables applications to continue operations for database sessions that can be reliably drained or migrated.

### Use Case for Application Continuity

In a typical case, a client has submitted a request to the database, which has built up both transactional and nontransactional states.

The state at the client remains current, potentially with entered data, returned data, and cached data and variables. However, the database session state, which the application needs to operate within, is lost.

If the client request has initiated one or more transactions, then the application is faced with the following possibilities:

- If a commit *has* been issued, then the commit message returned to the client is not durable. The client does not know whether the request committed, and where in the nontransactional processing state it reached.
- If a commit has *not* been issued, or if it was issued but did not execute, then the in-flight transaction is rolled back and must be resubmitted using a session in the correct state.

If the replay is successful, then database user service for planned and unplanned outages is not interrupted. If the database detects changes in the data seen and potentially acted on by the application, then the replay is rejected. Replay is not attempted when the time allowed for starting replay is exceeded, the application uses a restricted call, or the application has explicitly disabled replay using the `disableReplay` method.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about how Application Continuity works for database sessions

## Application Continuity for Planned Maintenance

Application Continuity for planned outages enables applications to continue operations for database sessions that can be reliably drained or migrated.

Scheduled maintenance need not disrupt application work. Application Continuity gives active work time to drain from its current location to a new location currently unaffected by maintenance. At the end of the drain interval, sessions may remain on the database instance where maintenance is planned. Instead of forcibly disconnecting these sessions, Application Continuity can fail over these sessions to a remaining site, and resubmit any in-flight transactions.

With Application Continuity enabled the database can do the following:

- Report no errors for either incoming or existing work during maintenance
- Redirect active database sessions to other functional services
- Rebalance database sessions, as needed, during and after the maintenance

Control the drain behavior during planned maintenance using the `drain_timeout` and `stop_option` service attributes of the SRVCTL utility, Global Data Services Control Utility (GDSCTL), and Oracle Data Guard Broker. The `DBMS_SERVICE` package provides the underlying infrastructure.

**See Also:**

- *Oracle Real Application Clusters Administration and Deployment Guide* to learn more Application Continuity
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_SERVICE`
- *Oracle Real Application Clusters Administration and Deployment Guide* for the SRVCTL command reference
- *Oracle Database Global Data Services Concepts and Administration Guide* for the GDSCTL command reference

## Application Continuity Architecture

The key components of Application Continuity are runtime, reconnection, and replay.

The phases are as follows:

1. Normal runtime

   In this phase, Application Continuity performs the following tasks:

   - Identifies database requests
   - Decides whether local and database calls are replayable
   - Builds proxy objects to enable replay, if necessary, and to manage queues
   - Holds original calls and validation of these calls until the end of the database request or replay is disabled

2. Reconnection

   This phase is triggered by a recoverable error. Application Continuity performs the following tasks:

   - Ensures that replay is enabled for the database requests
   - Manages timeouts
   - Obtains a new connection to the database, and then validates that this is a valid database target
   - Uses Transaction Guard to determine whether the last transaction committed successfully (committed transactions are not resubmitted)

3. Replay

   Application Continuity performs the following tasks:

   - Replays calls that are held in the queue
   - Disables replay if user-visible changes in results appear during the replay
   - Does not allow a commit, but does allow the last recall (which encountered the error) to commit

Following a successful replay, the request continues from the point of failure.

**See Also:**

- "Overview of Transaction Guard"
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about Application Continuity
- *Oracle Database JDBC Developer's Guide* to learn more about JDBC and Application Continuity

# Overview of Autonomous Transactions

An **autonomous transaction** is an independent transaction that can be called from another transaction, which is the main transaction. You can suspend the calling transaction, perform SQL operations and commit or undo them in the autonomous transaction, and then resume the calling transaction.

Autonomous transactions are useful for actions that must be performed independently, regardless of whether the calling transaction commits or rolls back. For example, in a stock purchase transaction, you want to commit customer data regardless of whether the overall stock purchase goes through. Additionally, you want to log error messages to a debug table even if the overall transaction rolls back.

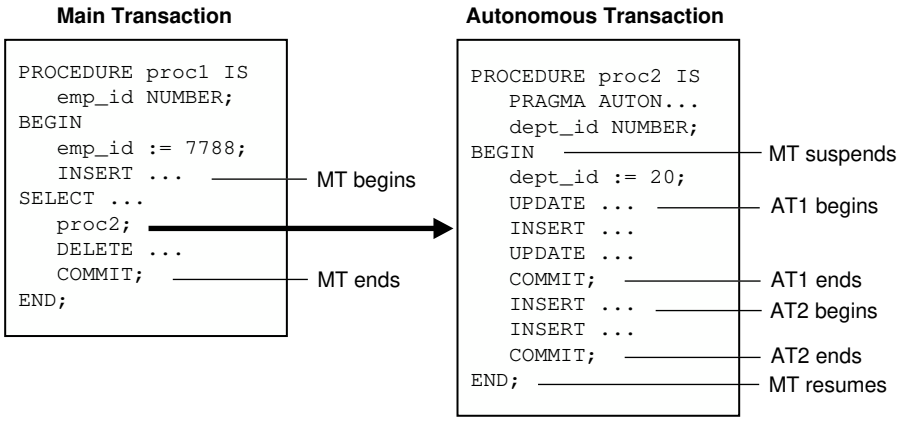
Autonomous transactions have the following characteristics:

- The autonomous transaction does not see uncommitted changes made by the main transaction and does not share locks or resources with the main transaction.
- Changes in an autonomous transaction are visible to other transactions upon commit of the autonomous transactions. Thus, users can access the updated information without having to wait for the main transaction to commit.
- Autonomous transactions can start other autonomous transactions. There are no limits, other than resource limits, on how many levels of autonomous transactions can be called.

In PL/SQL, an autonomous transaction executes within an *autonomous scope*, which is a routine marked with the pragma `AUTONOMOUS_TRANSACTION`. In this context, routines include top-level anonymous PL/SQL blocks and PL/SQL subprograms and triggers. A pragma is a directive that instructs the compiler to perform a compilation option. The pragma `AUTONOMOUS_TRANSACTION` instructs the database that this procedure, when executed, is to be executed as a new autonomous transaction that is independent of its parent transaction.

The following graphic shows how control flows from the main routine (MT) to an autonomous routine and back again. The main routine is `proc1` and the autonomous routine is `proc2`. The autonomous routine can commit multiple transactions (AT1 and AT2) before control returns to the main routine.

**Figure 12-4 Transaction Control Flow**




When you enter the executable section of an autonomous routine, the main routine suspends. When you exit the autonomous routine, the main routine resumes.

In Figure 12-4, the `COMMIT` inside `proc1` makes permanent not only its own work but any outstanding work performed in its session. However, a `COMMIT` in `proc2` makes permanent only the work performed in the `proc2` transaction. Thus, the `COMMIT` statements in transactions AT1 and AT2 have no effect on the MT transaction.

> **See Also:**
>
> *Oracle Database Development Guide* and *Oracle Database PL/SQL Language Reference* to learn how to use autonomous transactions

# Overview of Distributed Transactions

A **distributed transaction** is a transaction that includes one or more statements that update data on two or more distinct nodes of a distributed database, using a schema object called a database link.

A distributed database is a set of databases in a distributed system that can appear to applications as a single data source. A database link describes how one database instance can log in to another database instance.

Unlike a transaction on a local database, a distributed transaction alters data on multiple databases. Consequently, distributed transaction processing is more complicated because the database must coordinate the committing or rolling back of the changes in a transaction as an atomic unit. The entire transaction must commit or roll back. Oracle Database must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

- Two-Phase Commit
  The **two-phase commit mechanism** guarantees that *all* databases participating in a distributed transaction either all commit or all undo the statements in the transaction. The mechanism also protects implicit DML performed by integrity constraints, remote procedure calls, and triggers.

- In-Doubt Transactions
  An **in-doubt distributed transaction** occurs when a two-phase commit was interrupted by any type of system or network failure.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage distributed transactions

## Two-Phase Commit

The **two-phase commit mechanism** guarantees that *all* databases participating in a distributed transaction either all commit or all undo the statements in the transaction. The mechanism also protects implicit DML performed by integrity constraints, remote procedure calls, and triggers.

In a two-phase commit among multiple databases, one database coordinates the distributed transaction. The initiating node is called the *global coordinator*. The coordinator asks the other databases if they are prepared to commit. If any database responds with a no, then the entire transaction is rolled back. If all databases vote yes, then the coordinator broadcasts a message to make the commit permanent on each of the databases.

The two-phase commit mechanism is transparent to users who issue distributed transactions. In fact, users need not even know the transaction is distributed. A `COMMIT` statement denoting the end of a transaction automatically triggers the two-phase commit mechanism. No coding or complex statement syntax is required to include distributed transactions within the body of a database application.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about the two-phase commit mechanism
> - *Oracle Database SQL Language Reference*

## In-Doubt Transactions

An **in-doubt distributed transaction** occurs when a two-phase commit was interrupted by any type of system or network failure.

For example, two databases report to the coordinating database that they were prepared to commit, but the coordinating database instance fails immediately after receiving the messages. The two databases who are prepared to commit are now left hanging while they await notification of the outcome.

The recoverer (`RECO`) background process automatically resolves the outcome of in-doubt distributed transactions. After the failure is repaired and communication is reestablished, the `RECO` process of each local Oracle database automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

In the event of a long-term failure, Oracle Database enables each local administrator to manually commit or undo any distributed transactions that are in doubt because of the failure. This option enables the local database administrator to free any locked resources that are held indefinitely because of the long-term failure.

If a database must be recovered to a past time, then database recovery facilities enable database administrators at other sites to return their databases to the earlier point in time. This operation ensures that the global database remains consistent.

**See Also:**

- "Recoverer Process (RECO) "
- *Oracle Database Administrator's Guide* to learn how to manage in-doubt transactions

# Part V
# Oracle Database Storage Structures

This part describes the basic structural architecture of the Oracle database, including logical and physical storage structures.

This part contains the following chapters:

- Physical Storage Structures
- Logical Storage Structures

- Physical Storage Structures
  The physical database structures of an Oracle database are viewable at the operating system level.
- Logical Storage Structures
  This chapter describes the nature of and relationships among logical storage structures. These structures are created and recognized by Oracle Database and are not known to the operating system.

13

# Physical Storage Structures

The physical database structures of an Oracle database are viewable at the operating system level.

This chapter contains the following sections:


- Introduction to Physical Storage Structures
- Overview of Data Files
- Overview of Control Files
- Overview of the Online Redo Log


- Introduction to Physical Storage Structures
  One characteristic of an RDBMS is the independence of logical data structures such as tables, views, and indexes from physical storage structures such as files.
- Overview of Data Files
  At the operating system level, Oracle Database stores data in structures called **data files**. Every Oracle database, whether a CDB or PDB, must have at least one data file.
- Overview of Control Files
  The database **control file** is a small binary file associated with only one CDB.
- Overview of the Online Redo Log
  The most crucial structure for recovery is the **online redo log**, which consists of two or more preallocated files that store changes to the database as they occur. The online redo log records changes to the data files.

## Introduction to Physical Storage Structures

One characteristic of an RDBMS is the independence of logical data structures such as tables, views, and indexes from physical storage structures such as files.

Because physical and logical structures are separate, you can manage physical storage of data without affecting access to logical structures. For example, renaming a database file does not rename the tables stored in it.

An Oracle database is a set of files that store Oracle data in persistent storage. Starting in Oracle Database 21c, a multitenant container database (CDB) is the only supported architecture. A CDB contains one or more PDBs.

This section discusses the database files generated when you issue a `CREATE DATABASE` statement to create a CDB:

- Data files and temp files

  A data file is a physical file in persistent storage that was created by Oracle Database and contains data structures such as tables and indexes. A temp file is a data file that belongs to a temporary tablespace. The database writes data to these files in an Oracle proprietary format that cannot be read by other programs.

  When you create a PDB within a PDB, the PDB has its own set of data files within the overall set of data files that makes up the CDB. For example, you could create a CDB

named `mycdb`, and then create two PDBs within it: `hrpdb` and `salespdb`. In this case, `mycdb` would have its own set of data files and temp files, as would `hrpdb` and `salespdb`.

- Control files

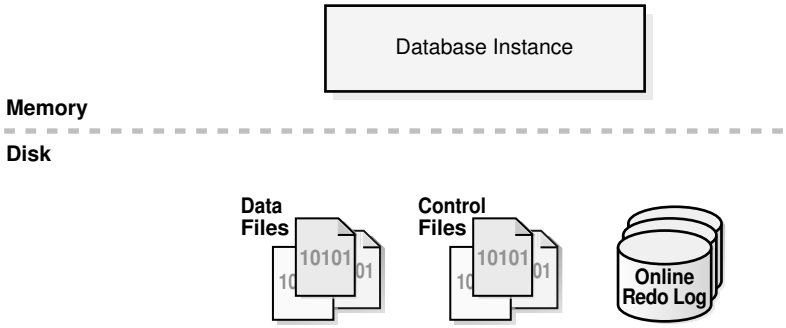
  A control file is a root file that tracks the physical components of the CDB. PDBs do not have their own separate control files.

- Online redo log files

  The online redo log is a set of files containing records of changes made to data within the CDB. PDBs do not have their own separate online redo log files.

A database instance is a set of memory structures that manage database files in a CDB. The following graphic shows the relationship between the instance and the files that it manages.

**Figure 13-1 Database Instance and Database Files**




- Mechanisms for Storing Database Files
  Several mechanisms are available for allocating and managing the storage of these files.
- Oracle Managed Files and User-Managed Files
  **Oracle Managed Files** is a file naming strategy that enables you to specify operations in terms of database objects rather than file names. For example, you can create a tablespace without specifying the names of its data files.
- Oracle Automatic Storage Management (Oracle ASM)
  **Oracle ASM** is a high-performance, ease-of-management storage solution for Oracle Database files. Oracle ASM is a volume manager and provides a file system designed exclusively for use by the database.
- Oracle Persistent Memory Filestore (PMEM Filestore)
  PMEM is byte-addressable persistent memory. Oracle Persistent Memory Filestore (PMEM Filestore) can store data files, online redo log files, and control files.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to create a database
- *Oracle Database SQL Language Reference* for `CREATE DATABASE` semantics and syntax

# Mechanisms for Storing Database Files

Several mechanisms are available for allocating and managing the storage of these files.

Mechanisms include:

- Operating system file system

  Most Oracle databases store files in a file system, which is a structure for storing and retrieving data.

  A file system enables storage space to be used for application data in a standard and consistent way. Each file has a name and appears as a contiguous address space to applications such as Oracle Database. The database can create, read, write, resize, and delete files.

  A file system is commonly built on top of a logical volume constructed by a software package called a logical volume manager (LVM). The LVM enables pieces of multiple physical disks to combine into what appears as a single contiguous address space.

- Cluster file system

  A cluster file system is a distributed file system that is a cluster of servers that collaborate to provide consistency and high performance to their clients. In an Oracle RAC environment, a cluster file system makes shared storage appear as a file system shared by many computers in a clustered environment. With a cluster file system, the failure of a computer in the cluster does not make the file system unavailable. In an operating system file system, however, if a computer sharing files through NFS or other means fails, then the file system is unavailable.

- Oracle Persistent Memory Filestore (PMEM Filestore)

  PMEM Filestore is designed to store database files in Persistent Memory (PMEM), which is byte-addressable persistent memory. You create a filestore using the `CREATE PMEM FILESTORE` command. At the Linux operating system level, the filestore stores data in a large ext4 file system mounted in `fsdax` mode. PMEM Filestore appears to Oracle Database as a native file system on Linux.

  > **Note:**
  >
  > Unlike a POSIX-compliant file system, PMEM Filestore cannot be shared by multiple Oracle databases.

- Oracle Automatic Storage Management (Oracle ASM)

  Oracle ASM includes a file system designed exclusively for use by Oracle Database.

A database can use a combination of the preceding storage mechanisms. For example, a database could store the server parameter file in a traditional file system, store database files in PMEM Filestore, and then archive the redo log files to a cluster file system.

> **See Also:**
>
> - "Oracle Automatic Storage Management (Oracle ASM)"
> - *Oracle Database Administrator's Guide* to view database storage structures by querying database views

## Oracle Managed Files and User-Managed Files

**Oracle Managed Files** is a file naming strategy that enables you to specify operations in terms of database objects rather than file names. For example, you can create a tablespace without specifying the names of its data files.

Oracle Managed Files eliminates the need for administrators to directly manage the operating system files in a database. Oracle ASM requires Oracle Managed Files.

> **Note:**
>
> This feature does not affect the creation or naming of administrative files such as trace files, audit files, and alert logs.

With user-managed files, you directly manage the operating system files in the database. You make the decisions regarding file structure and naming. For example, when you create a tablespace you set the name and path of the tablespace data files.

Through initialization parameters, you specify the file system directory for a specific type of file. The Oracle Managed Files feature ensures that the database creates a unique file and deletes it when no longer needed. The database internally uses standard file system interfaces to create and delete files for data files and temp files, control files, and recovery-related files stored in the fast recovery area.

Oracle Managed Files does not eliminate existing functionality. You can create new files while manually administering old files. Thus, a database can have a mixture of Oracle Managed Files and user-managed files.

> **See Also:**
>
> - "Overview of Diagnostic Files"
> - *Oracle Database Administrator's Guide* to learn how to use Oracle Managed Files

## Oracle Automatic Storage Management (Oracle ASM)

**Oracle ASM** is a high-performance, ease-of-management storage solution for Oracle Database files. Oracle ASM is a volume manager and provides a file system designed exclusively for use by the database.

Oracle ASM provides several advantages over conventional file systems and storage managers, including the following:

- Simplifies storage-related tasks such as creating and laying out databases and managing disk space
- Distributes data across physical disks to eliminate hot spots and to provide uniform performance across the disks
- Rebalances data automatically after storage configuration changes

To use Oracle ASM, you allocate partitioned disks for Oracle Database with preferences for striping and mirroring. Oracle ASM manages the disk space, distributing the I/O load across all available resources to optimize performance while removing the need for manual I/O tuning. For example, you can increase the size of the disk for the database or move parts of the database to new devices without having to shut down the database.

- Oracle ASM Storage Components
  Oracle Database can store a data file as an Oracle ASM file in an Oracle ASM disk group. Within a disk group, Oracle ASM exposes a file system interface for database files.
- Oracle ASM Instances
  An **Oracle ASM instance** is a special Oracle instance that manages Oracle ASM disks.

## Oracle ASM Storage Components

Oracle Database can store a data file as an Oracle ASM file in an Oracle ASM disk group. Within a disk group, Oracle ASM exposes a file system interface for database files.

The following figure shows the relationships between storage components in a database that uses Oracle ASM. The diagram depicts the relationship between an Oracle ASM file and a data file, although Oracle ASM can store other types of files. The crow's foot notation represents a one-to-many relationship.

**Figure 13-2 Oracle ASM Components**

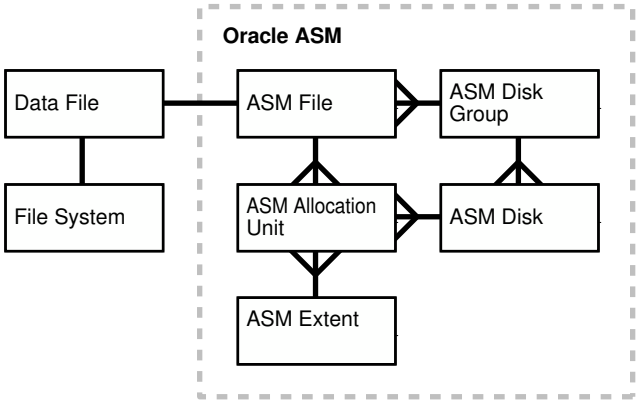



Figure 13-2 illustrates the following Oracle ASM concepts:

- Oracle ASM Disks

  An Oracle ASM disk is a storage device that is provisioned to an Oracle ASM disk group. An Oracle ASM disk can be a physical disk or partition, a Logical Unit Number (LUN) from a storage array, a logical volume, or a network-attached file.

Oracle ASM disks can be added or dropped from a disk group while the database is running. When you add a disk to a disk group, you either assign a disk name or the disk is given an Oracle ASM disk name automatically.

- Oracle ASM Disk Groups

  An Oracle ASM disk group is a collection of Oracle ASM disks managed as a logical unit. The data structures in a disk group are self-contained and consume some disk space in a disk group.

  Within a disk group, Oracle ASM exposes a file system interface for Oracle database files. The content of files that are stored in a disk group are evenly distributed, or striped, to eliminate hot spots and to provide uniform performance across the disks.

- Oracle ASM Files

  An Oracle ASM file is a file stored in an Oracle ASM disk group. Oracle Database communicates with Oracle ASM in terms of files. The database can store data files, control files, online redo log files, and other types of files as Oracle ASM files. When requested by the database, Oracle ASM creates an Oracle ASM file and assigns it a name beginning with a plus sign (+) followed by a disk group name, as in `+DISK1`.

  > **Note:**
  >
  > Oracle ASM files can coexist with other storage management options, such as third-party file systems. This capability simplifies the integration of Oracle ASM into pre-existing environments.

- Oracle ASM Extents

  An Oracle ASM extent is a section of an Oracle ASM file. An Oracle ASM file consists of one or more file extents. Each Oracle ASM extent consists of one or more allocation units on a specific disk.

  > **Note:**
  >
  > An Oracle ASM extent is different from the extent used to store data in a segment.

- Oracle ASM Allocation Units

  An Oracle ASM allocation unit is the fundamental unit of allocation within a disk group. An allocation unit is the smallest contiguous disk space that Oracle ASM allocates. One or more allocation units form an Oracle ASM extent.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* to learn more about Oracle ASM

## Oracle ASM Instances

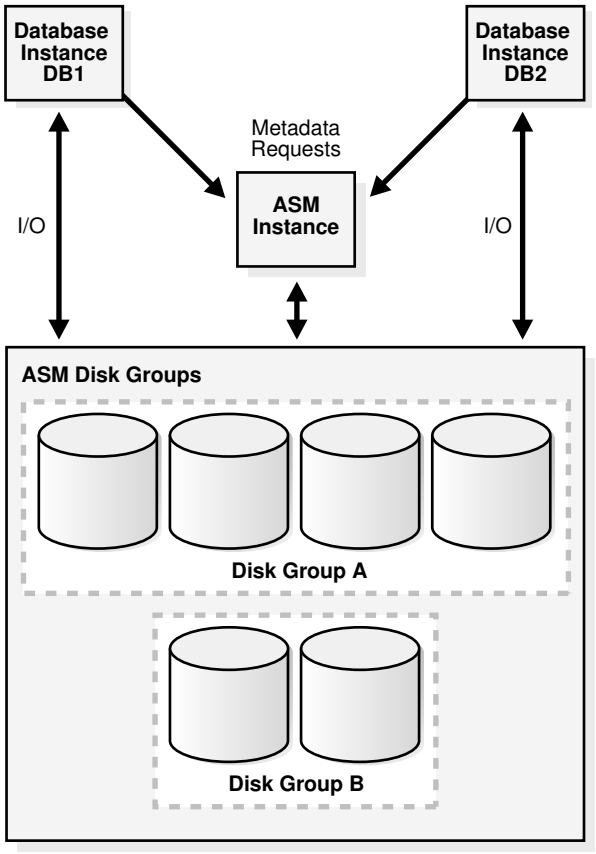
An **Oracle ASM instance** is a special Oracle instance that manages Oracle ASM disks.

Both the Oracle ASM and the database instances require shared access to the disks in an Oracle ASM disk group. Oracle ASM instances manage the metadata of the disk group and provide file layout information to the database instances. Database instances direct I/O to Oracle ASM disks without going through an Oracle ASM instance.

An Oracle ASM instance is built on the same technology as a database instance. For example, an Oracle ASM instance has a system global area (SGA) and background processes that are similar to those of a database instance. However, an Oracle ASM instance cannot mount a database and performs fewer tasks than a database instance.

Figure 13-3 shows a single-node configuration with one Oracle ASM instance and two database instances, each associated with a different single-instance database. The Oracle ASM instance manages the metadata and provides space allocation for the Oracle ASM files storing the data for the two databases. One Oracle ASM disk group has four Oracle ASM disks and the other has two disks. Both database instances can access the disk groups.

**Figure 13-3 Oracle ASM Instance and Database Instances**



> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* to learn more about Oracle ASM

## Oracle Persistent Memory Filestore (PMEM Filestore)

PMEM is byte-addressable persistent memory. Oracle Persistent Memory Filestore (PMEM Filestore) can store data files, online redo log files, and control files.

Queries can read bytes *directly* from PMEM without first migrating blocks into the database buffer cache. This high-performance mechanism avoids data redundancy and unnecessary I/O.

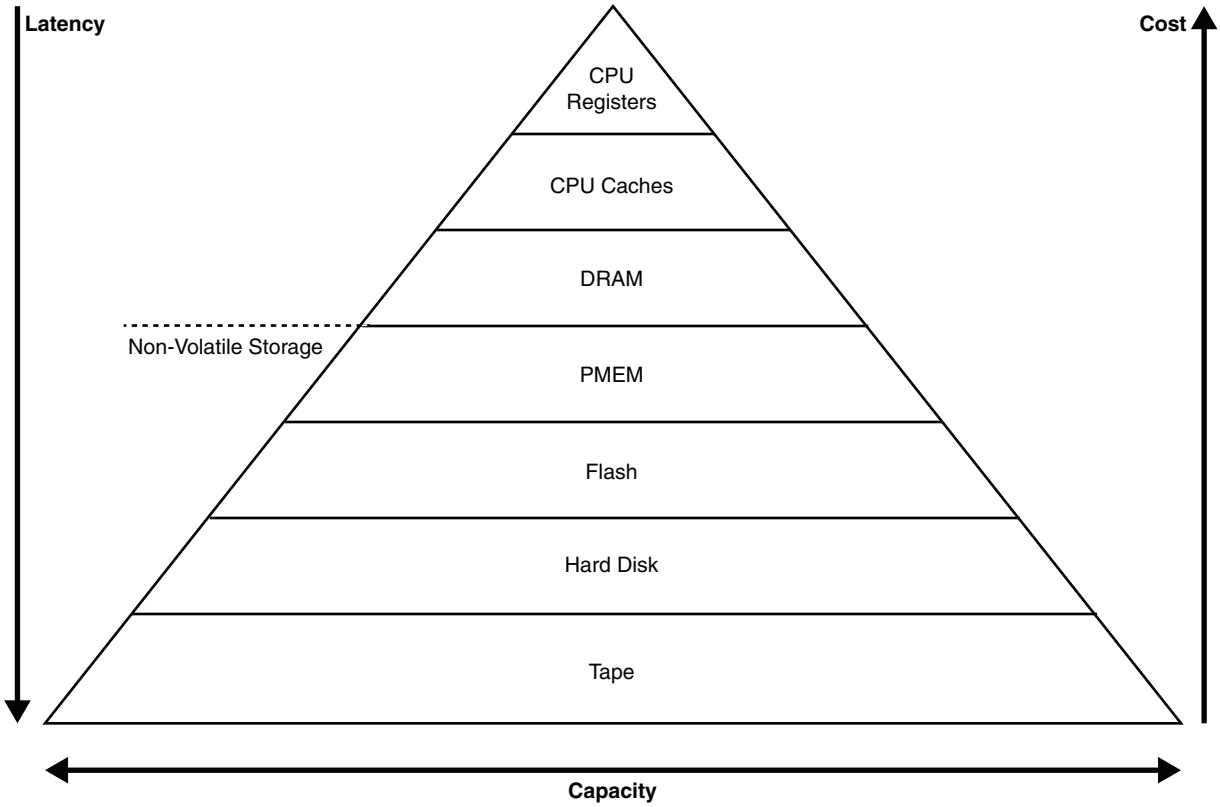
> **Note:**
>
> A PMEM filestore cannot store trace files, audit files, or server parameter files. The supported file types for PMEM are similar to the types for Oracle ASM.

- Directly Mapped Buffer Cache
  In the traditional storage pyramid, PMEM sits between DRAM, which is non-persistent, and Flash, which is persistent (non-volatile).
- File System Interface for PMEM Filestore
  PMEM Filestore implements a protocol called Filesystem in Userspace (FUSE).
- User Interface for PMEM Filestore
  To create a PMEM filestore, use the `CREATE PMEM FILESTORE` command.

## Directly Mapped Buffer Cache

In the traditional storage pyramid, PMEM sits between DRAM, which is non-persistent, and Flash, which is persistent (non-volatile).

**Figure 13-4 Storage Pyramid**




The database buffer cache resides in DRAM. Oracle processes have direct byte-addressable access to blocks in a PMEM filestore, which are mapped to the database buffer cache. The mapping mechanism is automatically invoked when you place a data file in a PMEM filestore.

Oracle Database uses an internal algorithm to decide whether to migrate data blocks from the PMEM filestore into DRAM. Typical reasons for migration include DML modifications, read consistency, and faster access for hot data blocks.

> **See Also:**
>
> "DRAM and PMEM Buffers"

## File System Interface for PMEM Filestore

PMEM Filestore implements a protocol called Filesystem in Userspace (FUSE).

FUSE enables non-privileged operating system users, such as the Oracle Database software owner, to create and manage file systems and the directories and files within them. For example, you can use the commands `cp`, `rm`, and `ls`.

A typical file system uses raw storage as its backing store, whereas PMEM Filestore gets storage from a native operating system file in a PMEM DAX file system. This file is called the **backing file** and is visible as a file in the operating system. PMEM Filestore subdivides the storage within the backing file and presents it as a local file system.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn more about PMEM Filestore

## User Interface for PMEM Filestore

To create a PMEM filestore, use the `CREATE PMEM FILESTORE` command.

The PMEM filestore is automatically mounted after it is created. The PMEM filestore appears under the specified mount point as if it were a native file system.

The `PMEM_FILESTORE` initialization parameter specifies a PMEM filestore that the database instance mounts automatically when it starts. The parameter is set to an ordered pair of strings. The first string in the parameter value list is the directory where PMEM Filestore is mounted. The second string is the backing file.

`V$PMEM_FILESTORE` shows metadata about a PMEM filestore. For example, you can view the directory path of the mount point, file path for the backing file, and the filestore size.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to create and manage a PMEM filestore

# Overview of Data Files

At the operating system level, Oracle Database stores data in structures called **data files**. Every Oracle database, whether a CDB or PDB, must have at least one data file.

- Use of Data Files
  Oracle Database physically stores tablespace data in data files.
- Permanent and Temporary Data Files
  A **permanent tablespace** contains persistent schema objects. Objects in permanent tablespaces are stored in data files.
- Online and Offline Data Files
  Every data file is either online (available) or offline (unavailable).
- Data File Structure
  Oracle Database creates a data file for a tablespace by allocating the specified amount of disk space plus the overhead for the data file header. The operating system under which Oracle Database runs is responsible for clearing old information and authorizations from a file before allocating it to the database.

## Use of Data Files

Oracle Database physically stores tablespace data in data files.

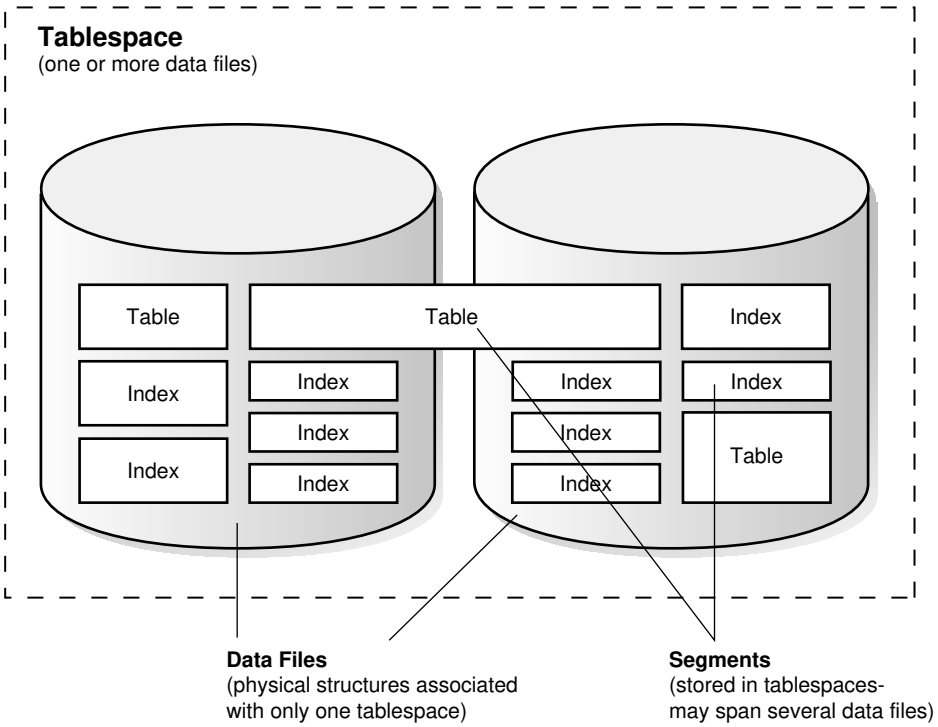
Each nonpartitioned **schema object** and each partition of an object is stored in its own **segment**, which belongs to only one tablespace. For example, the data for a nonpartitioned table is stored in a single segment, which in turn is stored in one tablespace. Tablespaces and data files are closely related, but have important differences:

- Each tablespace consists of one or more data files, which conform to the operating system in which Oracle Database is running.
- The data for a database is collectively stored in the data files located in each tablespace of the database.
- A segment can span one or more data files, but it cannot span multiple tablespaces.
- A CDB must have the `SYSTEM` and `SYSAUX` tablespaces. Each PDB within a CDB has its own `SYSTEM` and `SYSAUX` tablespaces. Oracle Database automatically allocates the first data files of any database for the `SYSTEM` tablespace during database creation.

  The `SYSTEM` tablespace contains the data dictionary, a set of tables that contains database metadata. Typically, a database also has an undo tablespace and a temporary tablespace (usually named `TEMP`).

The following figure shows the relationship between tablespaces, data files, and segments.

**Figure 13-5 Data Files and Tablespaces**




**See Also:**

- "Overview of Tablespaces"
- *Oracle Database Administrator's Guide* to learn how to manage data files

## Permanent and Temporary Data Files

A **permanent tablespace** contains persistent schema objects. Objects in permanent tablespaces are stored in data files.

A temporary tablespace contains schema objects only for the duration of a session. Locally managed temporary tablespaces have temporary files (temp files), which are special files designed to store data in hash, sort, and other operations. Temp files also store result set data when insufficient space exists in memory.

Temp files are similar to permanent data files, with the following exceptions:

- Permanent database objects such as tables are never stored in temp files.
- Temp files are always set to `NOLOGGING` mode, which means that they never have redo generated for them. Media recovery does not recognize temp files.
- You cannot make a temp file read-only.
- You cannot create a temp file with the `ALTER DATABASE` statement.
- When you create or resize temp files, they are not always guaranteed allocation of disk space for the file size specified. On file systems such as Linux and UNIX, temp files are created as *sparse files*. In this case, disk blocks are allocated not at file creation or resizing, but as the blocks are accessed for the first time.

> **Note:**
>
> Sparse files enable fast temp file creation and resizing; however, the disk could run out of space later when the temp files are accessed.

- Temp file information is shown in the data dictionary view `DBA_TEMP_FILES` and the dynamic performance view `V$TEMPFILE`, but not in `DBA_DATA_FILES` or the `V$DATAFILE` view.

> **See Also:**
>
> - "Temporary Tablespaces"
> - *Oracle Database Administrator's Guide* to learn how to manage temp files

## Online and Offline Data Files

Every data file is either online (available) or offline (unavailable).

You can alter the availability of individual data files or temp files by taking them offline or bringing them online. The database cannot access offline data files until they are brought online.

You may take data files offline for many reasons, including performing offline backups or block corruption. The database takes a data file offline automatically if the database cannot write to it.

Like a data file, a tablespace itself is offline or online. When you take a data file offline in an online tablespace, the tablespace itself remains online. You can make all data files of a tablespace temporarily unavailable by taking the tablespace itself offline.

Starting in Oracle Database 12c, you can use the `ALTER DATABASE MOVE DATAFILE` statement to move an online data file from one physical file to another while the database is open and accessing the file. You can use this technique to achieve the following goals:

- Move a tablespace from one kind of storage to another
- Move data files that are accessed infrequently to lower cost storage
- Make a tablespace read-only and move its data files to write-once storage, such as a write once read many (WORM) drive
- Move a database into Oracle ASM

> **See Also:**
>
> - "Online and Offline Tablespaces"
> - *Oracle Database Administrator's Guide* to learn how to alter data file availability
> - *Oracle Database Administrator's Guide* to learn how to move online data files
> - *Oracle Database SQL Language Reference* to learn about `ALTER DATABASE . . . MOVE DATAFILE`

## Data File Structure

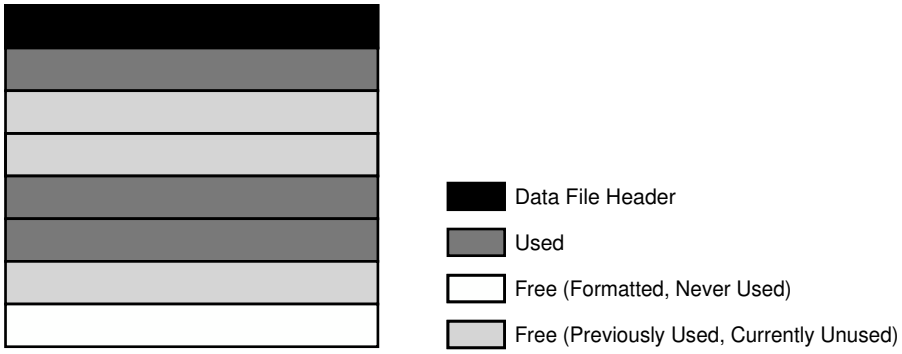
Oracle Database creates a data file for a tablespace by allocating the specified amount of disk space plus the overhead for the data file header. The operating system under which Oracle Database runs is responsible for clearing old information and authorizations from a file before allocating it to the database.

The data file header contains metadata about the data file such as its size and checkpoint SCN. Each header contains an absolute file number, which uniquely identifies the data file within the database, and a relative file number, which uniquely identifies a data file within a tablespace.

When Oracle Database first creates a data file, the allocated disk space is formatted but contains no user data. However, the database reserves the space to hold the data for future segments of the associated tablespace. As the data grows in a tablespace, Oracle Database uses the free space in the data files to allocate extents for the segment.

The following figure illustrates the different types of space in a data file. Extents are either used, which means they contain segment data, or free, which means they are available for reuse. Over time, updates and deletions of objects within a tablespace can create pockets of empty space that individually are not large enough to be reused for new data. This type of empty space is called *fragmented free space*.

**Figure 13-6 Space in a Data File**




> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to view data file information

# Overview of Control Files

The database **control file** is a small binary file associated with only one CDB.

Each CDB has one unique control file, although multiple identical copies are permitted. PDBs do not have separate control files.

- Use of Control Files
  Oracle Database uses the control file to locate database files and to manage the state of the database generally.
- Multiple Control Files
  Oracle Database enables multiple, identical control files to be open concurrently and written to the same database. By multiplexing a control file on different disks, the database can achieve redundancy and thereby avoid a single point of failure.
- Control File Structure
  Information about the database is stored in different sections of the control file. Each section is a set of records about an aspect of the database.

## Use of Control Files

Oracle Database uses the control file to locate database files and to manage the state of the database generally.

A control file contains information such as the following:

- The database name and database unique identifier (DBID)
- The time stamp of database creation
- Information about data files, online redo log files, and archived redo log files
- Tablespace information
- RMAN backups

The control file serves the following purposes:

- It contains information about data files, online redo log files, and so on that are required to open the database.

  The control file tracks structural changes to the database. For example, when an administrator adds, renames, or drops a data file or online redo log file, the database updates the control file to reflect this change.

- It contains metadata that must be accessible when the database is not open.

  For example, the control file contains information required to recover the database, including checkpoints. A checkpoint indicates the SCN in the redo stream where instance recovery would be required to begin. Every committed change before a checkpoint SCN is guaranteed to be saved on disk in the data files. At least every three seconds the checkpoint process records information in the control file about the checkpoint position in the online redo log.

Oracle Database reads and writes to the control file continuously during database use and must be available for writing whenever the database is open. For example, recovering a database involves reading from the control file the names of all the data files contained in the database. Other operations, such as adding a data file, update the information stored in the control file.

**See Also:**

- "Overview of Instance Recovery"
- "Checkpoint Process (CKPT)"
- *Oracle Database Administrator's Guide* to learn how to manage the control file

## Multiple Control Files

Oracle Database enables multiple, identical control files to be open concurrently and written to the same database. By multiplexing a control file on different disks, the database can achieve redundancy and thereby avoid a single point of failure.

**Note:**

Oracle recommends that you maintain multiple control file copies, each on a different disk.

If a control file becomes unusable, then the database instance fails when it attempts to access the damaged control file. When other current control file copies exist, then you can remount the database and open it without media recovery. If *all* control files of a database are lost, however, then the database instance fails and media recovery is required. Media recovery is not straightforward if an older backup of a control file must be used because a current copy is not available.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to maintain multiple control files
> - *Oracle Database Backup and Recovery User's Guide* to learn how to back up and restore control files

## Control File Structure

Information about the database is stored in different sections of the control file. Each section is a set of records about an aspect of the database.

For example, one section in the control file tracks data files and contains a set of records, one for each data file. Each section is stored in multiple logical control file blocks. Records can span blocks within a section.

The control file contains the following types of records:

- Circular reuse records

  A circular reuse record contains noncritical information that is eligible to be overwritten if needed. When all available record slots are full, the database either expands the control file to make room for a new record or overwrites the oldest record. Examples include records about archived redo log files and RMAN backups.

- Noncircular reuse records

  A noncircular reuse record contains critical information that does not change often and cannot be overwritten. Examples of information include tablespaces, data files, online redo log files, and redo threads. Oracle Database never reuses these records unless the corresponding object is dropped from the tablespace.

You can query the dynamic performance views, also known as `V$` views, to view the information stored in the control file. For example, you can query `V$DATABASE` to obtain the database name and DBID. However, only the database can modify the information in the control file.

Reading and writing the control file blocks is different from reading and writing data blocks. For the control file, Oracle Database reads and writes directly from the disk to the program global area (PGA). Each process allocates a certain amount of its PGA memory for control file blocks.

> **See Also:**
>
> - "Overview of the Dynamic Performance Views"
> - *Oracle Database Reference* to learn about the `V$CONTROLFILE_RECORD_SECTION` view
> - *Oracle Database Reference* to learn about the `CONTROL_FILE_RECORD_KEEP_TIME` initialization parameter

# Overview of the Online Redo Log

The most crucial structure for recovery is the **online redo log**, which consists of two or more preallocated files that store changes to the database as they occur. The online redo log records changes to the data files.

- Use of the Online Redo Log
  The database maintains online redo log files to protect against data loss. Specifically, after an instance failure, the online redo log files enable Oracle Database to recover committed data that it has not yet written to the data files.
- How Oracle Database Writes to the Online Redo Log
  The online redo log for a database instance is called a **redo thread**.
- Structure of the Online Redo Log
  Online redo log files contain redo records.

## Use of the Online Redo Log

The database maintains online redo log files to protect against data loss. Specifically, after an instance failure, the online redo log files enable Oracle Database to recover committed data that it has not yet written to the data files.

Server processes write every transaction synchronously to the redo log buffer, which the LGWR process then writes to the online redo log. Contents of the online redo log include uncommitted transactions, and schema and object management statements.

As the database makes changes to the undo segments, the database also writes these changes to the online redo logs. Consequently, the online redo log always contains the undo data for permanent objects. You can configure the database to store all undo data for temporary objects in a temporary undo segment, which saves space and improves performance, or allow the database to store both permanent and temporary undo data in the online redo log.

Oracle Database uses the online redo log only for recovery. However, administrators can query online redo log files through a SQL interface in the Oracle LogMiner utility. Redo log files are a useful source of historical information about database activity.

**See Also:**

- "Overview of Instance Recovery"
- "Temporary Undo Segments"
- "Process Architecture" to learn about Oracle processes
- *Oracle Database Administrator's Guide* to learn about temporary undo segments
- *Oracle Database Reference* to learn about the `TEMP_UNDO_ENABLED` initialization parameter

## How Oracle Database Writes to the Online Redo Log

The online redo log for a database instance is called a **redo thread**.

In single-instance configurations, only one instance accesses a database, so only one redo thread is present. In an Oracle Real Application Clusters (Oracle RAC) configuration, however, multiple instances concurrently access a database, with each instance having its own redo thread. A separate redo thread for each instance avoids contention for a single set of online redo log files.

An online redo log consists of two or more online redo log files. Oracle Database requires a minimum of two files to guarantee that one file is always available for writing in case the other file is in the process of being cleared or archived.

- Online Redo Log Switches
  Oracle Database uses only one online redo log file at a time to store records written from the redo log buffer.
- Multiple Copies of Online Redo Log Files
  Oracle Database can automatically maintain two or more identical copies of the online redo log in separate locations.
- Archived Redo Log Files
  An **archived redo log file** is a copy of a filled member of an online redo log group.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* to learn about online redo log groups in Oracle RAC

## Online Redo Log Switches

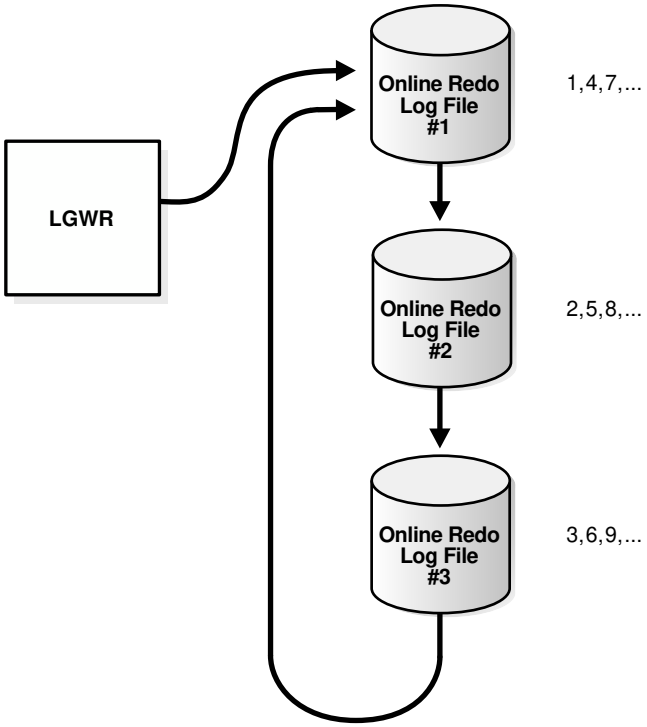
Oracle Database uses only one online redo log file at a time to store records written from the redo log buffer.

The online redo log file to which the log writer process (LGWR) process is actively writing is called the current online redo log file.

A log switch occurs when the database stops writing to one online redo log file and begins writing to another. Normally, a switch occurs when the current online redo log file is full and writing must continue. However, you can configure log switches to occur at regular intervals, regardless of whether the current online redo log file is filled, and force log switches manually.

Log writer writes to online redo log files circularly. When log writer fills the last available online redo log file, the process writes to the first log file, restarting the cycle. Figure 13-7 illustrates the circular writing of the redo log.

**Figure 13-7 Reuse of Online Redo Log Files**




The numbers in Figure 13-7 shows the sequence in which LGWR writes to each online redo log file. The database assigns each file a new log sequence number when a log switches and log writers begins writing to it. When the database reuses an online redo log file, this file receives the next available log sequence number.

Filled online redo log files are available for reuse depending on the archiving mode:

- If archiving is disabled, which means that the database is in `NOARCHIVELOG` mode, then a filled online redo log file is available after the changes recorded in it have been checkpointed (written) to disk by database writer (DBW).
- If archiving is enabled, which means that the database is in ARCHIVELOG mode, then a filled online redo log file is available to log writer after the changes have been written to the data files *and* the file has been archived.

In some circumstances, log writer may be prevented from reusing an existing online redo log file. An active online redo log file is required for instance recovery, where as an inactive online redo log file is not required for instance recovery. Also, an online redo log file may be in the process of being cleared.

**See Also:**

- "Overview of Background Processes"
- *Oracle Database Administrator's Guide* to learn how to manage the online redo log

# Multiple Copies of Online Redo Log Files

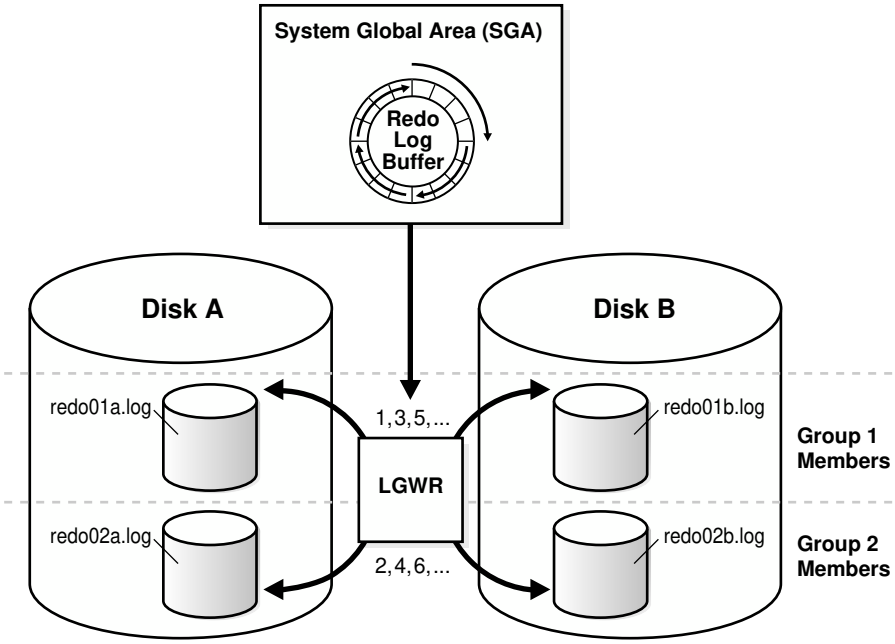
Oracle Database can automatically maintain two or more identical copies of the online redo log in separate locations.

An online redo log group consists of an online redo log file and its redundant copies. Each identical copy is a member of the online redo log group. Each group is defined by a number, such as group 1, group 2, and so on.

Maintaining multiple members of an online redo log group protects against the loss of the redo log. Ideally, the locations of the members should be on separate disks so that the failure of one disk does not cause the loss of the entire online redo log.

In Figure 13-8, `A_LOG1` and `B_LOG1` are identical members of group 1, while `A_LOG2` and `B_LOG2` are identical members of group 2. Each member in a group must be the same size. LGWR writes concurrently to group 1 (members `A_LOG1` and `B_LOG1`), then writes concurrently to group 2 (members `A_LOG2` and `B_LOG2`), then writes to group 1, and so on. LGWR never writes concurrently to members of different groups.

**Figure 13-8 Multiple Copies of Online Redo Log Files**




> **Note:**
>
> Oracle recommends that you multiplex the online redo log. The loss of log files can be catastrophic if recovery is required. When you multiplex the online redo log, the database must increase the amount of I/O it performs. Depending on your system, this additional I/O may impact overall database performance.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to maintain multiple copies of the online redo log files
> - *Oracle Data Guard Concepts and Administration* to learn about the automated transfer of redo data between members of a Data Guard configuration

## Archived Redo Log Files

An **archived redo log file** is a copy of a filled member of an online redo log group.

An archived redo log file is not considered part of the database. Rather, it is an offline copy of an online redo log file created by the database and written to a user-specified location.

Archived redo log files are a crucial part of a backup and recovery strategy. You can use archived redo log files to:

- Recover a database backup
- Update a standby database
- Obtain information about the history of a database using the Oracle LogMiner utility

The operation of generating an archived redo log file is known as archiving. This operation is either automatic or manual. It is only possible when the database is in `ARCHIVELOG` mode.

An archived redo log file includes the redo entries and the log sequence number of the identical member of the online redo log group. In "Multiple Copies of Online Redo Log Files", files `A_LOG1` and `B_LOG1` are identical members of Group 1. If the database is in `ARCHIVELOG` mode, and if automatic archiving is enabled, then the archiver process (ARCn) will archive one of these files. If `A_LOG1` is corrupted, then the process can archive `B_LOG1`. The archived redo log contains a copy of every redo log group that existed starting from the time that you enabled archiving.

> **See Also:**
>
> - *Oracle Database Utilities* to learn about Oracle LogMiner
> - *Oracle Database Administrator's Guide* to learn how to manage the archived redo log
> - *Oracle Data Guard Concepts and Administration* to learn how to configure standby redo log archival

# Structure of the Online Redo Log

Online redo log files contain redo records.

A redo record is made up of a group of change vectors, each of which describes a change to a data block. For example, an update to a salary in the `employees` table generates a redo record that describes changes to the data segment block for the table, the undo segment data block, and the transaction table of the undo segments.

The redo records have all relevant metadata for the change, including the following:

- SCN and time stamp of the change
- Transaction ID of the transaction that generated the change
- SCN and time stamp when the transaction committed (if it committed)
- Type of operation that made the change
- Name and type of the modified data segment

> **See Also:**
>
> "Overview of Data Blocks"

14

# Logical Storage Structures

This chapter describes the nature of and relationships among logical storage structures. These structures are created and recognized by Oracle Database and are not known to the operating system.

This chapter contains the following sections:


- Introduction to Logical Storage Structures
- Overview of Data Blocks
- Overview of Extents
- Overview of Segments
- Overview of Tablespaces


- Introduction to Logical Storage Structures
  Oracle Database allocates logical space for all data in the database.
- Overview of Data Blocks
  Oracle Database manages the logical storage space in the data files of a database in a unit called a **data block**, also called an *Oracle block* or *page*. A data block is the minimum unit of database I/O.
- Overview of Extents
  An extent is a unit of database storage made up of logically contiguous data blocks. Data blocks can be physically spread out on disk because of RAID striping and file system implementations.
- Overview of Segments
  A segment is a set of extents that contains all the data for a logical storage structure within a tablespace.
- Overview of Tablespaces
  A **tablespace** is a logical storage container for segments.

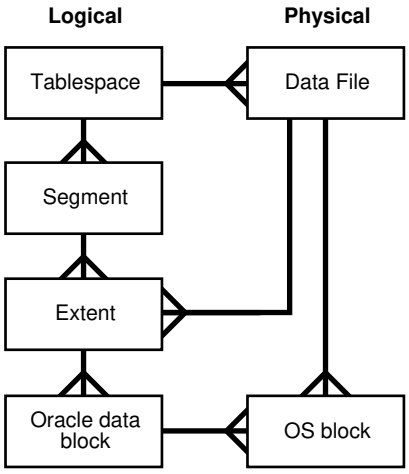
## Introduction to Logical Storage Structures

Oracle Database allocates logical space for all data in the database.

The logical units of database space allocation are data blocks, extents, segments, and tablespaces. At a physical level, the data is stored in data files on disk. The data in the data files is stored in operating system blocks.

The following figure is an entity-relationship diagram for physical and logical storage. The crow's foot notation represents a one-to-many relationship.

**Figure 14-1 Logical and Physical Storage**




- Logical Storage Hierarchy
  A segment contains one or more extents, each of which contains multiple data blocks.
- Logical Space Management
  Oracle Database must use logical space management to track and allocate the extents in a tablespace.

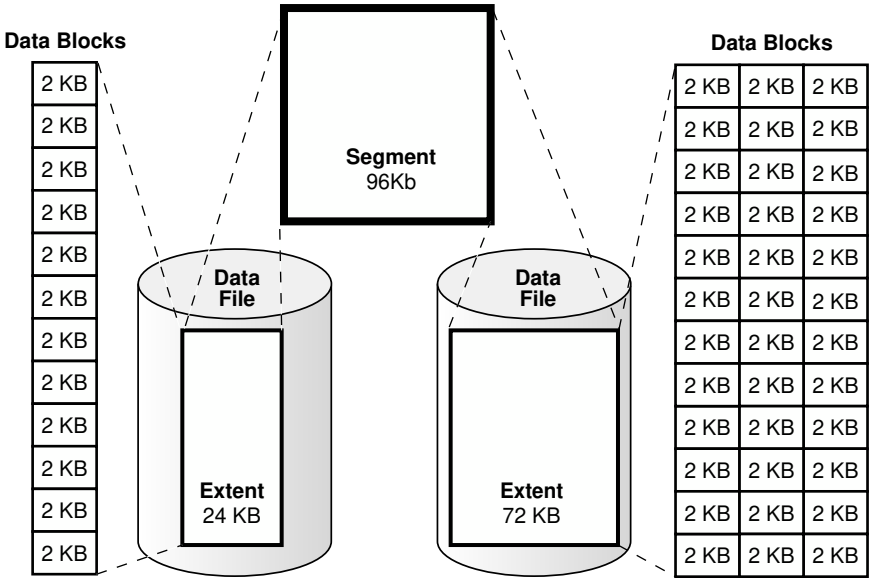
> **See Also:**
>
> "Physical Storage Structures"

## Logical Storage Hierarchy

A segment contains one or more extents, each of which contains multiple data blocks.

The following figure shows the relationships among data blocks, extents, and segments within a tablespace. In this example, a segment has two extents stored in different data files.

**Figure 14-2 Segments, Extents, and Data Blocks Within a Tablespace**




From the lowest level of granularity to the highest, Oracle Database stores data as follows:

- A data block is the smallest logical unit of data storage in Oracle Database.

  One logical data block corresponds to a specific number of bytes of persistent storage, for example, 2 KB. Data blocks are the smallest units of storage that Oracle Database can use or allocate.

  Traditionally, data files have been stored on magnetic disk or Solid State Disk (SSD). Oracle Database also supports Persistent Memory (PMEM) for storage of database files. Regardless of how the underlying data is physically stored, Oracle processes always read and write logical data blocks.

- An extent is a set of logically contiguous data blocks allocated for storing a specific type of information

  In the preceding graphic, the 24 KB extent has 12 data blocks, while the 72 KB extent has 36 data blocks.

- A segment is a set of extents allocated for a specific database object, such as a table.

  For example, the data for the `employees` table is stored in its own data segment, whereas each index for `employees` is stored in its own index segment. Every database object that consumes storage consists of a single segment.

- A tablespace is a database storage unit that contains one or more segments.

  Each segment belongs to one and only one tablespace. Thus, all extents for a segment are stored in the same tablespace. Within a tablespace, a segment can include extents from multiple data files, as shown in the preceding graphic. For example, one extent for a segment may be stored in `users01.dbf`, while another is stored in `users02.dbf`. A single extent can never span data files.

> **See Also:**
>
> "Introduction to Physical Storage Structures"

## Logical Space Management

Oracle Database must use logical space management to track and allocate the extents in a tablespace.

When a database object requires an extent, the database must have a method of finding and providing it. Similarly, when an object no longer requires an extent, the database must have a method of making the free extent available.

Oracle Database manages space within a tablespace based on the type that you create. You can create either of the following types of tablespaces:

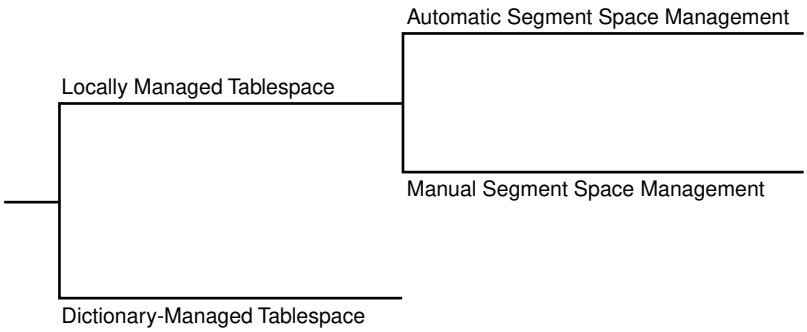
- Locally managed tablespaces (default)

  The database uses bitmaps in the tablespaces themselves to manage extents. Thus, locally managed tablespaces have a part of the tablespace set aside for a bitmap. Within a tablespace, the database can manage segments with automatic segment space management (ASSM) or manual segment space management (MSSM).
- Dictionary-managed tablespaces

  The database uses the data dictionary to manage extents.

Figure 14-3 shows the alternatives for logical space management in a tablespace.

**Figure 14-3 Logical Space Management**




- Locally Managed Tablespaces
  A locally managed tablespace maintains a bitmap in the data file header to track free and used space in the data file body.
- Dictionary-Managed Tablespaces
  A dictionary-managed tablespace uses the data dictionary to manage its extents.

> **See Also:**
>
> "Overview of the Data Dictionary"

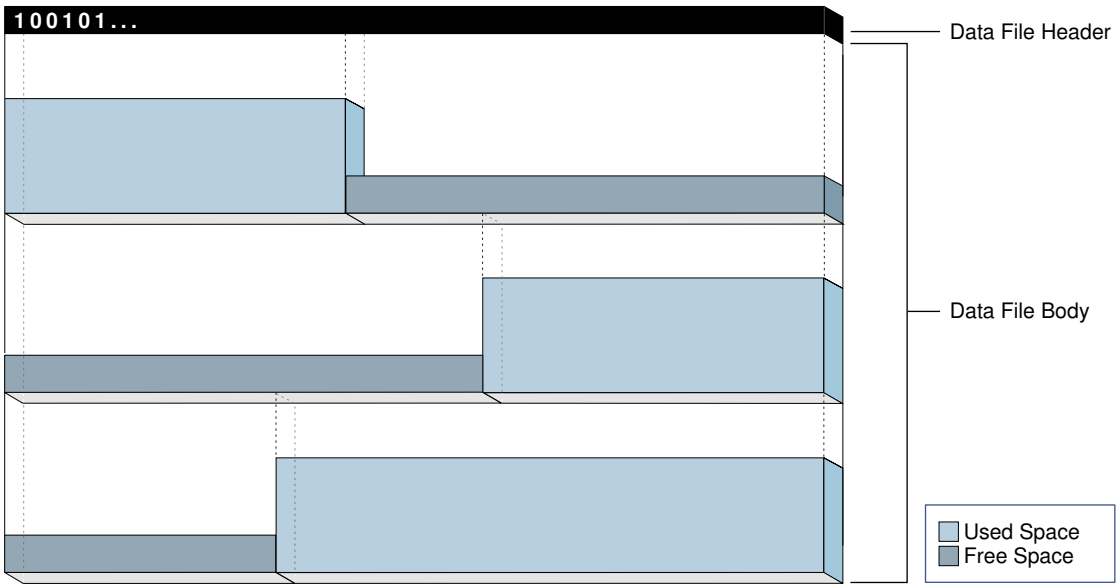
## Locally Managed Tablespaces

A locally managed tablespace maintains a bitmap in the data file header to track free and used space in the data file body.

Each bit corresponds to a group of blocks. When space is allocated or freed, Oracle Database changes the bitmap values to reflect the new status of the blocks.

The following graphic is a conceptual representation of bitmap-managed storage. A `1` in the header refers to used space, whereas a `0` refers to free space.

**Figure 14-4 Bitmap-Managed Storage**




A locally managed tablespace has the following advantages:

- Avoids using the data dictionary to manage extents

  Recursive operations can occur in dictionary-managed tablespaces if consuming or releasing space in an extent results in another operation that consumes or releases space in a data dictionary table or undo segment.
- Tracks adjacent free space automatically

  In this way, the database eliminates the need to coalesce free extents.
- Determines the size of locally managed extents automatically

  Alternatively, all extents can have the same size in a locally managed tablespace and override object storage options.

> **Note:**
>
> Oracle strongly recommends the use of locally managed tablespaces with Automatic Segment Space Management.

Segment space management is an attribute inherited from the tablespace that contains the segment. Within a locally managed tablespace, the database can manage segments automatically or manually. For example, segments in tablespace `users` can be managed automatically while segments in tablespace `tools` are managed manually.

- Automatic Segment Space Management
  The **automatic segment space management (ASSM)** method uses bitmaps to manage space in a tablespace.
- Manual Segment Space Management
  The legacy **manual segment space management (MSSM)** method uses a linked list called a **free list** to manage free space in the segment.

## Automatic Segment Space Management

The **automatic segment space management (ASSM)** method uses bitmaps to manage space in a tablespace.

Bitmaps provide the following advantages:

- Simplified administration

  ASSM avoids the need to manually determine correct settings for many storage parameters. Only one crucial SQL parameter controls space allocation: `PCTFREE`. This parameter specifies the percentage of space to be reserved in a block for future updates (see "Percentage of Free Space in Data Blocks").
- Increased concurrency

  Multiple transactions can search separate lists of free data blocks, thereby reducing contention and waits. For many standard workloads, application performance with ASSM is better than the performance of a well-tuned application that uses MSSM.
- Dynamic affinity of space to instances in an Oracle Real Application Clusters (Oracle RAC) environment

ASSM is more efficient and is the default for permanent, locally managed tablespaces.

> **Note:**
>
> This chapter assumes the use of ASSM in all of its discussions of logical storage space.

## Manual Segment Space Management

The legacy **manual segment space management (MSSM)** method uses a linked list called a **free list** to manage free space in the segment.

For a database object that has free space, a free list keeps track of blocks under the high water mark (HWM) (HWM), which is the dividing line between segment space that is used and not yet used. As blocks are used, the database puts blocks on or removes blocks from the free list as needed.

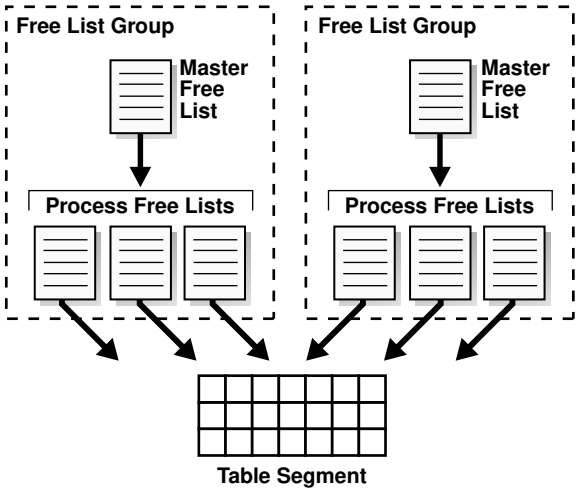
In addition to `PCTFREE`, MSSM requires you to control space allocation with SQL parameters such as `PCTUSED`, `FREELISTS`, and `FREELIST GROUPS`. `PCTUSED` sets the percentage of free space that must exist in a currently used block for the database to put it on the free list. For example, if you set `PCTUSED` to `40` in a `CREATE TABLE` statement, then you cannot insert rows into a block in the segment until less than 40% of the block space is used.

For example, suppose you insert a row into a table. The database checks a free list of the table for the first available block. If the row does not fit in the block, and if the used space in the block is greater than or equal to `PCTUSED`, then the database removes the block from the list and searches for another block. If you delete rows from the block, then the database checks whether used space in the block is now less than `PCTUSED`. If so, then the database places the block at the beginning of the free list.

An object may have multiple free lists. In this way, multiple sessions performing DML on a table can use different lists, which can reduce contention. Each database session uses only one free list for the duration of its session.

As shown in Figure 14-5, you can also create an object with one or more *free list groups*, which are collections of free lists. Each group has a *master free list* that manages the individual *process free list* in the group. Space overhead for free lists, especially for free list groups, can be significant.

**Figure 14-5 Free List Groups**




Managing segment space manually can be complex. You must adjust `PCTFREE` and `PCTUSED` to reduce row migration and avoid wasting space. For example, if every used block in a segment is half full, and if `PCTUSED` is `40`, then the database does not permit inserts into any of these blocks. Because of the difficulty of fine-tuning space allocation parameters, Oracle strongly recommends ASSM. In ASSM, `PCTFREE` determines whether a new row can be inserted into a block, but it does not use free lists and ignores `PCTUSED`.

**See Also:**

- "Chained and Migrated Rows "
- *Oracle Database Administrator's Guide* to learn about locally managed tablespaces
- *Oracle Database Administrator's Guide* to learn more about automatic segment space management
- *Oracle Database SQL Language Reference* to learn about storage parameters such as `PCTFREE` and `PCTUSED`

## Dictionary-Managed Tablespaces

A dictionary-managed tablespace uses the data dictionary to manage its extents.

Oracle Database updates tables in the data dictionary whenever an extent is allocated or freed for reuse. For example, when a table needs an extent, the database queries the data dictionary tables, and searches for free extents. If the database finds space, then it modifies one data dictionary table and inserts a row into another. In this way, the database manages space by modifying and moving data.

The SQL that the database executes in the background to obtain space for database objects is recursive SQL. Frequent use of recursive SQL can have a negative impact on performance because updates to the data dictionary must be serialized. Locally managed tablespaces, which are the default, avoid this performance problem.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to migrate tablespaces from dictionary-managed to locally managed

# Overview of Data Blocks

Oracle Database manages the logical storage space in the data files of a database in a unit called a **data block**, also called an *Oracle block* or *page*. A data block is the minimum unit of database I/O.

- Data Blocks and Operating System Blocks
  At the physical level, database data is stored in disk files made up of operating system blocks.
- Data Block Format
  Every data block has a format or internal structure that enables the database to track the data and free space in the block. This format is similar whether the data block contains table, index, or table cluster data.
- Data Block Compression
  The database can use **table compression** to eliminate duplicate values in a data block. This section describes the format of data blocks that use compression.
- Space Management in Data Blocks
  As the database fills a data block from the bottom up, the amount of free space between the row data and the block header decreases.
- Overview of Index Blocks
  An **index block** is a special type of data block that manages space differently from table blocks. Oracle Database uses index blocks to manage the logical storage space in an index.

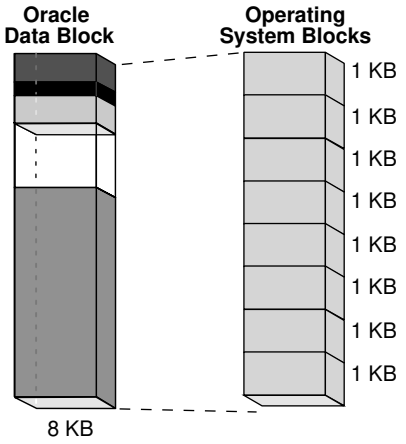
## Data Blocks and Operating System Blocks

At the physical level, database data is stored in disk files made up of operating system blocks.

An operating system block is the minimum unit of data that the operating system can read or write. In contrast, an Oracle block is a logical storage structure whose size and structure are not known to the operating system.

The following figure shows that operating system blocks may differ in size from data blocks. The database requests data in multiples of data blocks, not operating system blocks.

**Figure 14-6 Data Blocks and Operating System Blocks**




When the database requests a data block, the operating system translates this operation into a requests for data in permanent storage. The logical separation of data blocks from operating system blocks has the following implications:

- Applications do not need to determine the physical addresses of data on disk.
- Database data can be striped or mirrored on multiple physical disks.
- Database Block Size
  Every database has a database block size.
- Tablespace Block Size
  You can create individual tablespaces whose block size differs from the `DB_BLOCK_SIZE` setting.

## Database Block Size

Every database has a database block size.

The `DB_BLOCK_SIZE` initialization parameter sets the data block size for a database when it is created. The size is set for the `SYSTEM` and `SYSAUX` tablespaces and is the default for all other tablespaces. The database block size cannot be changed except by re-creating the database.

If `DB_BLOCK_SIZE` is not set, then the default data block size is operating system-specific. The standard data block size for a database is 4 KB or 8 KB. If the size differs for data blocks and operating system blocks, then the data block size must be a multiple of the operating system block size.

> **See Also:**
>
> - *Oracle Database Reference* to learn about the `DB_BLOCK_SIZE` initialization parameter
> - *Oracle Database Administrator's Guide* and *Oracle Database Performance Tuning Guide* to learn how to choose block sizes

## Tablespace Block Size

You can create individual tablespaces whose block size differs from the `DB_BLOCK_SIZE` setting.

A nonstandard block size can be useful when moving a transportable tablespace to a different platform.

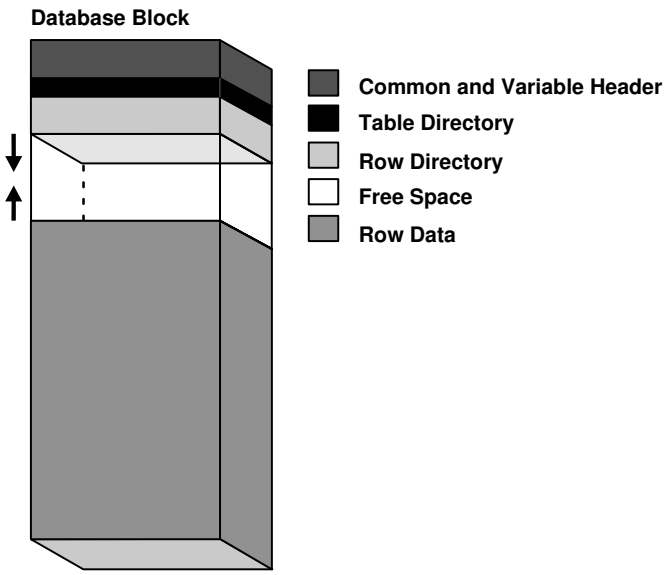
> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to specify a nonstandard block size for a tablespace

# Data Block Format

Every data block has a format or internal structure that enables the database to track the data and free space in the block. This format is similar whether the data block contains table, index, or table cluster data.

The following figure shows the format of an uncompressed data block.

**Figure 14-7 Data Block Format**



- Data Block Overhead
  Oracle Database uses the **block overhead** to manage the block itself. The block overhead is not available to store user data.
- Row Format
  The row data part of the block contains the actual data, such as table rows or index key entries. Just as every data block has an internal format, every row has a row format that enables the database to track the data in the row.

> **See Also:**
>
> "Data Block Compression" to learn about compressed blocks

## Data Block Overhead

Oracle Database uses the **block overhead** to manage the block itself. The block overhead is not available to store user data.

As shown in "Data Block Format", the block overhead includes the following parts:

- Block header

  This part contains general information about the block, including disk address and segment type. For blocks that are transaction-managed, the block header contains active and historical transaction information.

  A transaction entry is required for every transaction that updates the block. Oracle Database initially reserves space in the block header for transaction entries. In data blocks allocated to segments that support transactional changes, free space can also hold transaction entries when the header space is depleted. The space required for transaction entries is operating system dependent. However, transaction entries in most operating systems require approximately 23 bytes.
- Table directory

  For a heap-organized table, this directory contains metadata about tables whose rows are stored in this block. In a table cluster, multiple tables can store rows in the same block.
- Row directory

  For a heap-organized table, this directory describes the location of rows in the data portion of the block. The database can place a row anywhere in the bottom of the block. The row address is recorded in one of the slots of the row directory vector.

  A rowid points to a specific file, block, and row number. For example, in the rowid `AAAPecAAFAAAABSAAA`, the final `AAA` represents the row number. The row number is an index into an entry in the row directory. The row directory entry contains a pointer to the location of the row on the data block. If the database moves a row within a block, then the database updates the row directory entry to modify the pointer. The rowid stays constant.

  After the database allocates space in the row directory, the database does not reclaim this space after deleting rows. Thus, a block that is currently empty but formerly had up to 50 rows continues to have 100 bytes allocated for the row directory. The database reuses this space only when a session inserts new rows in the block.

Some parts of the block overhead are fixed in size, but the total size is variable. On average, the block overhead totals 84 to 107 bytes.

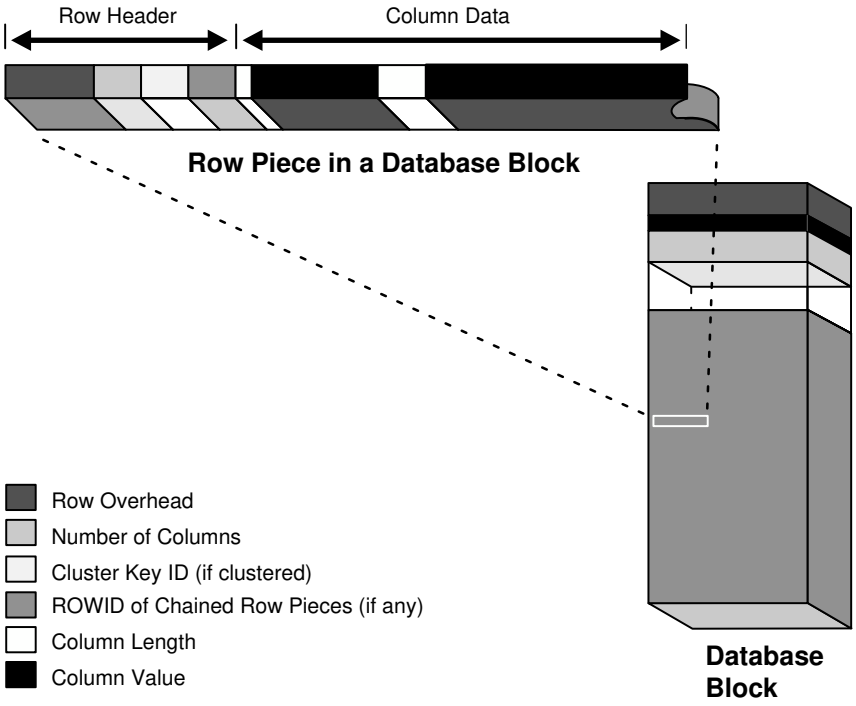
## Row Format

The row data part of the block contains the actual data, such as table rows or index key entries. Just as every data block has an internal format, every row has a row format that enables the database to track the data in the row.

Oracle Database stores rows as variable-length records. A row is contained in one or more sections. Each section is called a row piece. Each row piece has a row header and column data.

The following figure shows the format of a row.

**Figure 14-8 The Format of a Row Piece**




- Row Header
  Oracle Database uses the row header to manage the row piece stored in the block.
- Column Data
  After the row header, the column data section stores the actual data in the row. The row piece usually stores columns in the order listed in the `CREATE TABLE` statement, but this order is not guaranteed. For example, columns of type `LONG` are created last.
- Rowid Format
  Oracle Database uses a **rowid** to uniquely identify a row. Internally, the rowid is a structure that holds information that the database needs to access a row. A rowid is not physically stored in the database, but is inferred from the file and block on which the data is stored.

## Row Header

Oracle Database uses the row header to manage the row piece stored in the block.

The row header contains information such as the following:

- Columns in the row piece
- Pieces of the row located in other data blocks

  If an entire row can be inserted into a single data block, then Oracle Database stores the row as one row piece. However, if all of the row data cannot be inserted into a single block or an update causes an existing row to outgrow its block, then the database stores the row in multiple row pieces. A data block usually contains only one row piece per row.
- Cluster keys for table clusters

A row fully contained in one block has at least 3 bytes of row header.

> **See Also:**
>
> - "Chained and Migrated Rows "
> - "Overview of Table Clusters"

## Column Data

After the row header, the column data section stores the actual data in the row. The row piece usually stores columns in the order listed in the `CREATE TABLE` statement, but this order is not guaranteed. For example, columns of type `LONG` are created last.

As shown in the figure in "Row Format", for each column in a row piece, Oracle Database stores the column length and data separately. The space required depends on the data type. If the data type of a column is variable length, then the space required to hold a value can grow and shrink with updates to the data.

Each row has a slot in the row directory of the data block header. The slot points to the beginning of the row.

> **See Also:**
>
> "Table Storage" and "Index Storage"

## Rowid Format

Oracle Database uses a **rowid** to uniquely identify a row. Internally, the rowid is a structure that holds information that the database needs to access a row. A rowid is not physically stored in the database, but is inferred from the file and block on which the data is stored.

An extended rowid includes a data object number. This rowid type uses a base 64 encoding of the physical address for each row. The encoding characters are `A-Z`, `a-z`, `0-9`, `+`, and `/`.

**Example 14-1 ROWID Pseudocolumn**

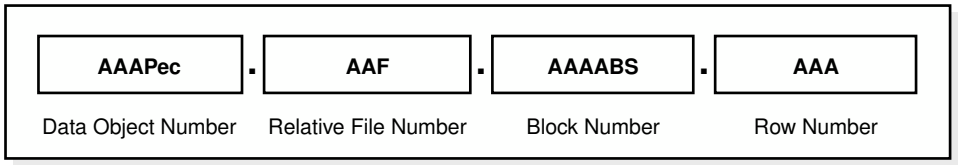
The following example queries the `ROWID` pseudocolumn to show the extended rowid of the row in the `employees` table for employee 100:

```
SQL> SELECT ROWID FROM employees WHERE employee_id = 100;
```

```
ROWID
------------------
AAAPecAAFAAAABSAAA
```

The following figure illustrates the format of an extended rowid.

**Figure 14-9 ROWID Format**




An extended rowid is displayed in a four-piece format, `OOOOOOFFFBBBBBBRRR`, with the format divided into the following components:

- `OOOOOO`

  The data object number identifies the segment (data object `AAAPec` in the sample query). A data object number is assigned to every database segment. Schema objects in the same segment, such as a table cluster, have the same data object number.

- `FFF`

  The tablespace-relative data file number identifies the data file that contains the row (file `AAF` in the sample query).

- `BBBBBB`

  The data block number identifies the block that contains the row (block `AAAABS` in the sample query). Block numbers are relative to their data file, not their tablespace. Thus, two rows with identical block numbers could reside in different data files of the same tablespace.

- `RRR`

  The row number identifies the row in the block (row `AAA` in the sample query).

After a rowid is assigned to a row piece, the rowid can change in special circumstances. For example, if row movement is enabled, then the rowid can change because of partition key updates, Flashback Table operations, shrink table operations, and so on. If row movement is disabled, then a rowid can change if the row is exported and imported using Oracle Database utilities.

> **Note:**
>
> Internally, the database performs row movement as if the row were physically deleted and reinserted. However, row movement is considered an update, which has implications for triggers.

> **See Also:**
>
> - "Rowid Data Types"
> - *Oracle Database SQL Language Reference* to learn about rowids

## Data Block Compression

The database can use **table compression** to eliminate duplicate values in a data block. This section describes the format of data blocks that use compression.

The format of a data block that uses basic table and advanced row compression is essentially the same as an uncompressed block. The difference is that a symbol table at the beginning of the block stores duplicate values for the rows and columns. The database replaces occurrences of these values with a short reference to the symbol table.

**Example 14-2 Format of Compressed Data Blocks**

Assume that the following rows are stored in a data block for the seven-column `sales` table:

```
2190,13770,25-NOV-00,S,9999,23,161
2225,15720,28-NOV-00,S,9999,25,1450
34005,120760,29-NOV-00,P,9999,44,2376
9425,4750,29-NOV-00,I,9999,11,979
1675,46750,29-NOV-00,S,9999,19,1121
```

When basic table or advanced row compression is applied to this table, the database replaces duplicate values with a symbol reference. The following conceptual representation of the compression shows the symbol `*` replacing `29-NOV-00` and `%` replacing `9999`:

```
2190,13770,25-NOV-00,S,%,23,161
2225,15720,28-NOV-00,S,%,25,1450
34005,120760,*,P,%,44,2376
9425,4750,*,I,%,11,979
1675,46750,*,S,%,19,1121
```

Table 14-1 conceptually represents the symbol table that maps symbols to values.

**Table 14-1 Symbol Table**

| Symbol | Value | Column | Rows |
|---|---|---|---|
| * | 29-NOV-00 | 3 | 958-960 |
| % | 9999 | 5 | 956-960 |

> **See Also:**
>
> "Table Compression"

# Space Management in Data Blocks

As the database fills a data block from the bottom up, the amount of free space between the row data and the block header decreases.

Free space in a data block can also shrink during updates, as when changing a trailing null value to a non-null value. The database manages free space in the data block to optimize performance and avoid wasted space.

> **Note:**
>
> This section assumes the use of automatic segment space management.

- Percentage of Free Space in Data Blocks
  The `PCTFREE` SQL parameter sets the minimum percentage of a data block reserved as free space for updates to existing rows. `PCTFREE` is important for preventing row migration and avoiding wasted space.
- Optimization of Free Space in Data Blocks
  While the percentage of free space cannot be *less* than `PCTFREE`, the amount of free space can be *greater*. For example, setting `PCTFREE` to 20% prevents the total amount of free space from dropping to 5% of the block, but allows 50% of the block to be free.
- Chained and Migrated Rows
  Oracle Database uses chaining and migration to manage rows that are too large to fit into a single block.

## Percentage of Free Space in Data Blocks

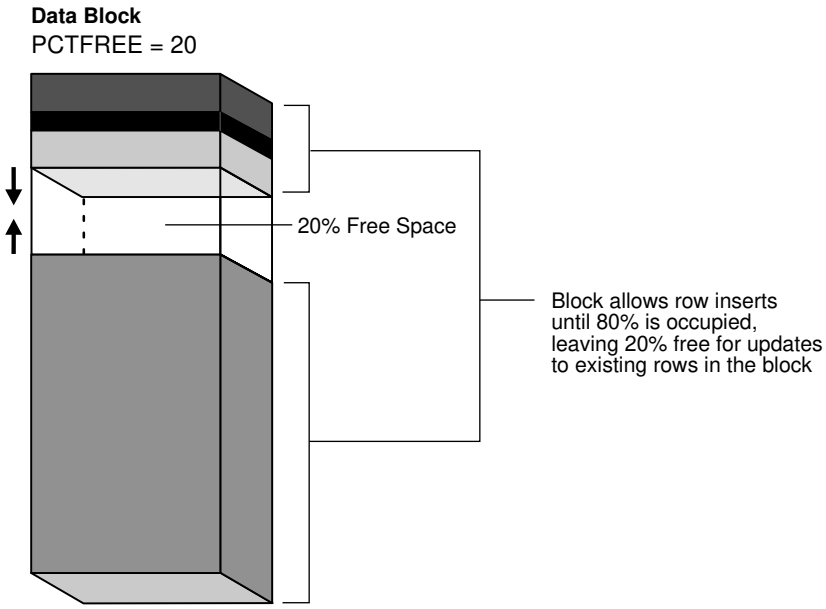
The `PCTFREE` SQL parameter sets the minimum percentage of a data block reserved as free space for updates to existing rows. `PCTFREE` is important for preventing row migration and avoiding wasted space.

For example, assume that you create a table that will require only occasional updates, most of which will not increase the size of the existing data. You specify the `PCTFREE` parameter within a `CREATE TABLE` statement as follows:

```
CREATE TABLE test_table (n NUMBER) PCTFREE 20;
```

Figure 14-10 shows how a `PCTFREE` setting of `20` affects space management. The database adds rows to the block over time, causing the row data to grow upwards toward the block header, which is itself expanding downward toward the row data. The `PCTFREE` setting ensures that *at least* 20% of the data block is free. For example, the database prevents an `INSERT` statement from filling the block so that the row data and header occupy a combined 90% of the total block space, leaving only 10% free.

**Figure 14-10 PCTFREE**




**Note:**

This discussion does not apply to LOB data types, which do not use the `PCTFREE` storage parameter or free lists.

**See Also:**

- *Oracle Database SecureFiles and Large Objects Developer's Guide*
- *Oracle Database SQL Language Reference* for the syntax and semantics of the `PCTFREE` parameter

## Optimization of Free Space in Data Blocks

While the percentage of free space cannot be *less* than `PCTFREE`, the amount of free space can be *greater*. For example, setting `PCTFREE` to 20% prevents the total amount of free space from dropping to 5% of the block, but allows 50% of the block to be free.

- Optimization by Increasing Free Space
  Some DML statements can increase free space in data blocks.
- Optimization by Coalescing Fragmented Space
  Released space may or may not be contiguous with the main area of free space in a data block. Noncontiguous free space is called *fragmented space*.

## Optimization by Increasing Free Space

Some DML statements can increase free space in data blocks.

The following statements can increase space:

- `DELETE` statements
- `UPDATE` statements that either update existing values to smaller values or increase existing values and force a row to migrate
- `INSERT` statements on a table that uses advanced row compression

  If `INSERT` statements fill a block with data, then the database invokes block compression, which may result in the block having more free space.

The space released is available for `INSERT` statements under the following conditions:

- If the `INSERT` statement is in the same transaction, and if it is after the statement that frees space, then the statement can use the space.
- If the `INSERT` statement is in a separate transaction from the statement that frees space (perhaps run by another user), and if space is needed, then the statement can use the space made available, but only after the other transaction commits.

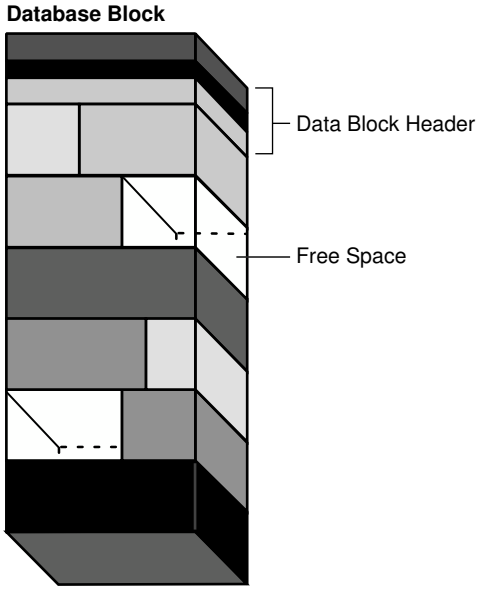
> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn about advanced row compression

## Optimization by Coalescing Fragmented Space

Released space may or may not be contiguous with the main area of free space in a data block. Noncontiguous free space is called *fragmented space*.

The following figure shows a data block with noncontiguous free space.

**Figure 14-11 Data Block with Fragmented Space**

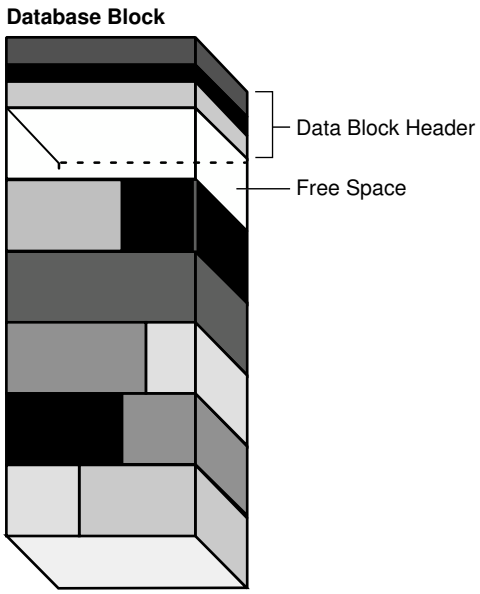



Oracle Database automatically and transparently coalesces the free space of a data block *only* when the following conditions are true:

- An `INSERT` or `UPDATE` statement attempts to use a block that contains sufficient free space to contain a new row piece.
- The free space is fragmented so that the row piece cannot be inserted in a contiguous section of the block.

After coalescing, the amount of free space is identical to the amount before the operation, but the space is now contiguous. Figure 14-12 shows a data block after space has been coalesced.

**Figure 14-12 Data Block After Coalescing Free Space**



Oracle Database performs coalescing only in the preceding situations because otherwise performance would decrease because of the continuous coalescing of the free space in data blocks.

## Chained and Migrated Rows

Oracle Database uses chaining and migration to manage rows that are too large to fit into a single block.

> **Note:**
>
> A chained row in a standard table is different from a row chain in a blockchain table. Oracle Database uses different technology to manage rows in a blockchain table.

The following situations are possible:

- The row is too large to fit into one data block when it is first inserted.

  In row chaining, Oracle Database stores the data for the row in a chain of one or more data blocks reserved for the segment. Row chaining most often occurs with large rows. Examples include rows that contain a column of data type `LONG` or `LONG RAW` , or a row with a huge number of columns. Row chaining in these cases is unavoidable.

- A row that originally fit into one data block is updated so that the overall row length increases, but insufficient free space exists to hold the updated row.

  In row migration, Oracle Database moves the entire row to a new data block, assuming the row can fit in a new block. The original row piece of a migrated row contains a pointer or "forwarding address" to the new block containing the migrated row. The rowid of a migrated row does not change.

- A row has more than 255 columns.

  Oracle Database can only store 255 columns in a row piece. Thus, if you insert a row into a table that has 1000 columns, then the database creates 4 row pieces, typically chained over multiple blocks.

Figure 14-13 depicts the insertion of a large row in a data block. The row is too large for the left block, so the database chains the row by placing the first row piece in the left block and the second row piece in the right block.

**Figure 14-13 Row Chaining**

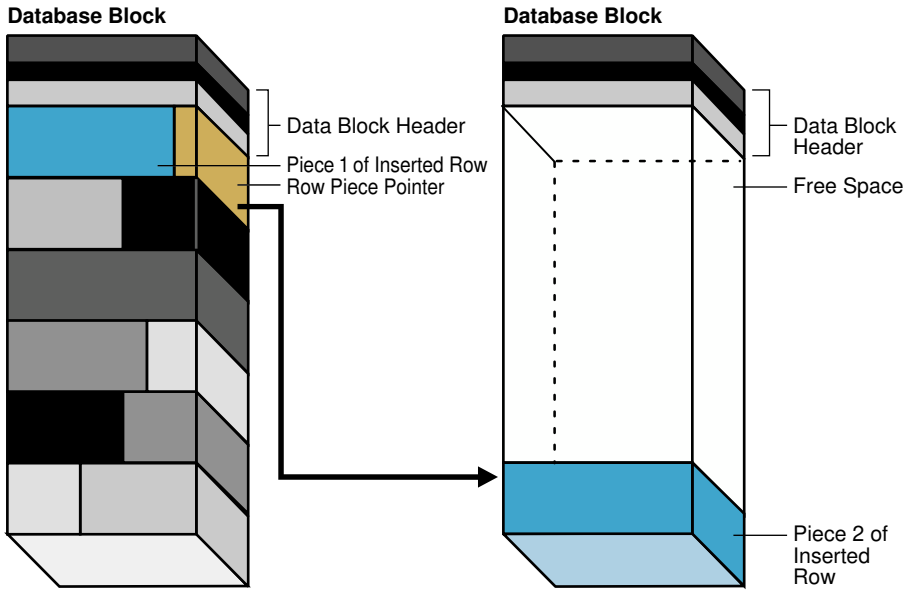
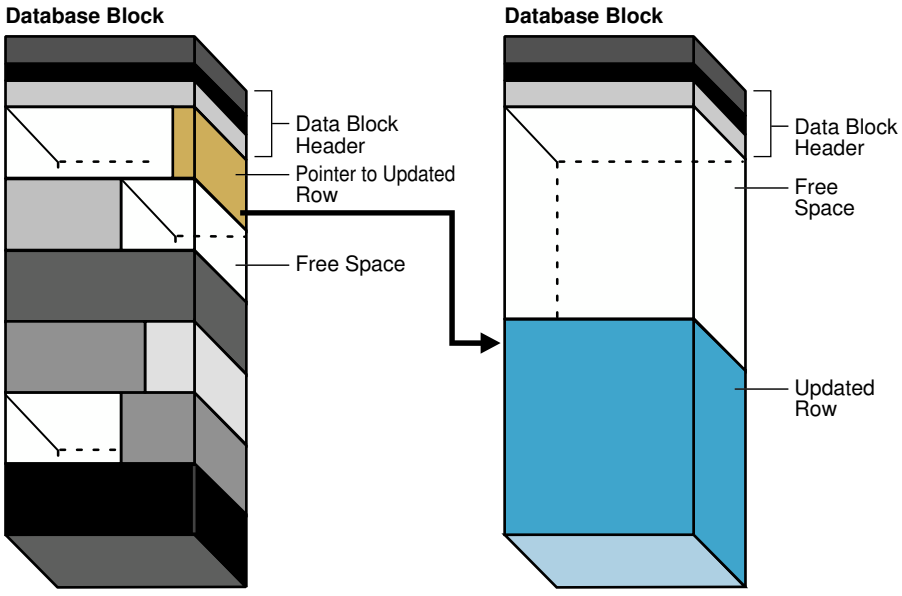



Figure 14-14, the left block contains a row that is updated so that the row is now too large for the block. The database moves the entire row to the right block and leaves a pointer to the migrated row in the left block.

**Figure 14-14 Row Migration**




When a row is chained or migrated, the I/O needed to retrieve the data increases. This situation results because Oracle Database must scan multiple blocks to retrieve the information for the row. For example, if the database performs one I/O to read an index and one I/O to read a nonmigrated table row, then an additional I/O is required to obtain the data for a migrated row.

The Segment Advisor, which can be run both manually and automatically, is an Oracle Database component that identifies segments that have space available for reclamation. The advisor can offer advice about objects that have significant free space or too many chained rows.

**See Also:**

- "Row Storage" and "Rowids of Row Pieces"
- "Overview of Blockchain Tables"
- *Oracle Database Administrator's Guide* to learn how to reclaim wasted space
- *Oracle Database Performance Tuning Guide* to learn about reducing chained and migrated rows

# Overview of Index Blocks

An **index block** is a special type of data block that manages space differently from table blocks. Oracle Database uses index blocks to manage the logical storage space in an index.

- Types of Index Blocks
  An index contains a root block, branch blocks, and leaf blocks.
- Storage of Index Entries
  Index entries are stored in index blocks in the same way as table rows in a data block The index entries in the block portion are not stored in binary order, but in a heap.
- Reuse of Slots in an Index Block
  The database can reuse space within an index block.
- Coalescing an Index Block
  Index coalescing compacts existing index data in place and, if the reorganization frees blocks, leaves the free blocks in the index structure. Thus, coalescing does not release index blocks for other uses or cause the index to reallocate blocks.

## Types of Index Blocks

An index contains a root block, branch blocks, and leaf blocks.

The block types are defined as follows:

- Root block

  This block identifies the entry point into the index.
- Branch blocks

  The databases navigates through branch blocks when searching for an index key.
- Leaf blocks

  These blocks contain the indexed key values rowids that point to the associated rows. The leaf blocks store key values in sorted order so that the database can search efficiently for all rows in a range of key values.

## Storage of Index Entries

Index entries are stored in index blocks in the same way as table rows in a data block The index entries in the block portion are not stored in binary order, but in a heap.

The database manages the row directory in an index block differently from the directory in a data block. The entries in the row directory (not the entries in the body of the index block) are ordered by key value. For example, in the row directory, the directory entry for index key `000000` precedes the directory entry for index key `111111`, and so on.

The ordering of entries in the row directory improves the efficiency of index scans. In a range scan, the database must read all the index keys specified in the range. The database traverses the branch blocks to identify the leaf block that contains the first key. Because entries in the row directory are sorted, the database can use a binary search to find the first index key in the range, and then progress sequentially through the entries in the row directory until it finds the last key. In this way, the database avoids reading all the keys in the leaf block body.

> **See Also:**
>
> "Data Block Overhead"

## Reuse of Slots in an Index Block

The database can reuse space within an index block.

For example, an application may insert a value into a column and then delete the value. When a row requires space, the database can reuse the index slot formerly occupied by the deleted value.

An index block usually has many more rows than a heap-organized table block. The ability to store many rows in a single index block makes it easier for the database to maintain an index because it avoids frequent splits of the block to store new data.

An index cannot coalesce itself, although you can manually coalesce it with an `ALTER INDEX` statement with the `REBUILD` or `COALESCE` options. For example, if you populate a column with values `1` to `500000`, and if you then delete the rows that contain even numbers, then the index will contain 250,000 empty slots. The database reuses a slot only if it can insert data that fits into an index block that contains an empty slot.

## Coalescing an Index Block

Index coalescing compacts existing index data in place and, if the reorganization frees blocks, leaves the free blocks in the index structure. Thus, coalescing does not release index blocks for other uses or cause the index to reallocate blocks.

Oracle Database does not automatically compact the index: you must run an `ALTER INDEX` statement with the `REBUILD` or `COALESCE` options.

Figure 14-15 shows an index of the `employees.department_id` column before the index is coalesced. The first three leaf blocks are only partially full, as indicated by the gray fill lines.

**Figure 14-15 Index Before Coalescing**

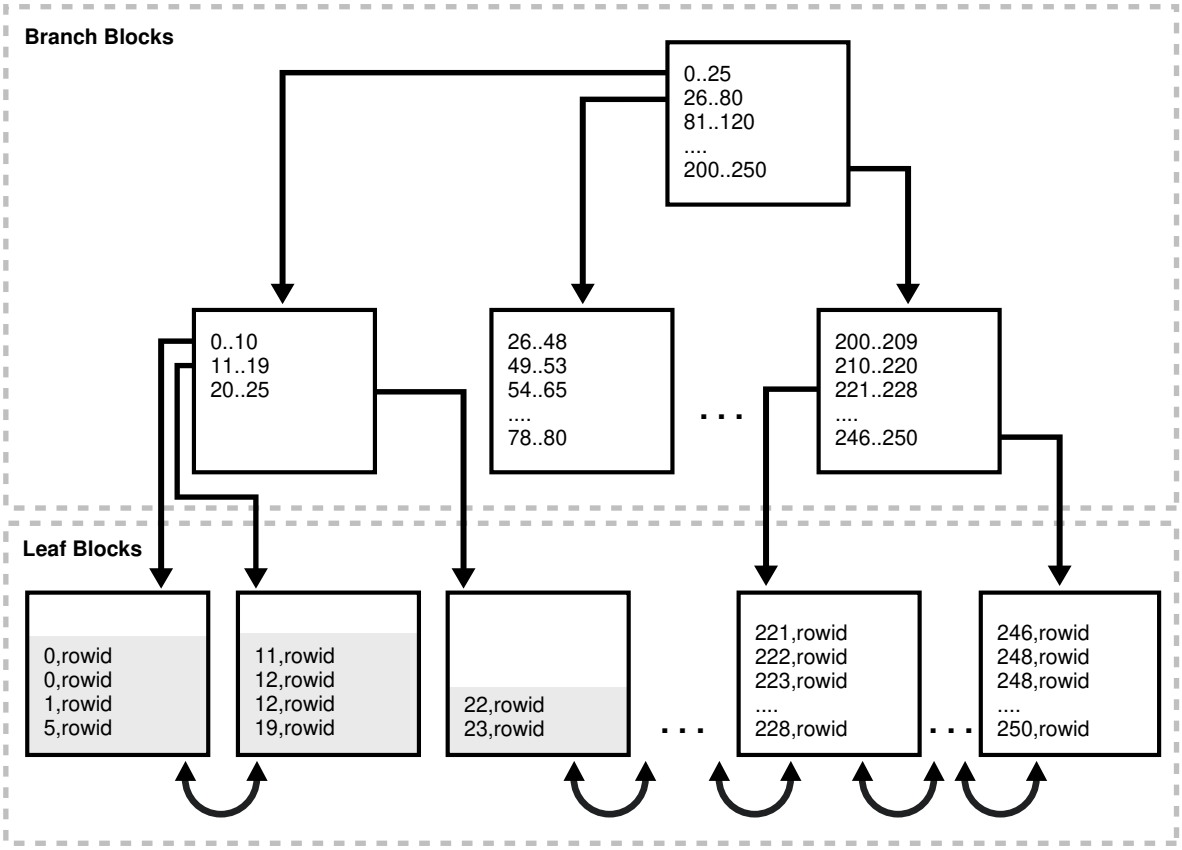
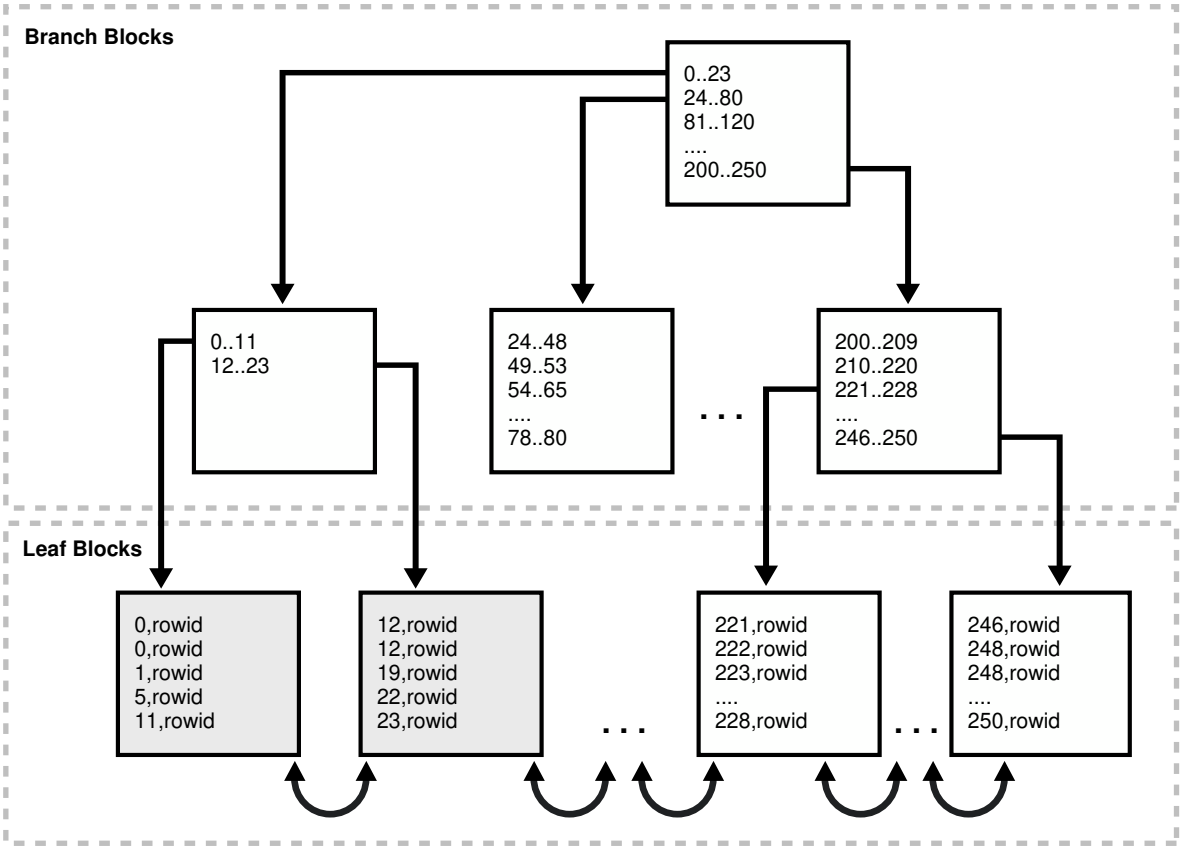



Figure 14-16 shows the index in Figure 14-15 after the index has been coalesced. The first two leaf blocks are now full, as indicated by the gray fill lines, and the third leaf block has been freed.

**Figure 14-16 Index After Coalescing**




**See Also:**

- *Oracle Database Administrator's Guide* to learn how to coalesce and rebuild indexes
- *Oracle Database SQL Language Reference* to learn about the `COALESCE` statement

# Overview of Extents

An extent is a unit of database storage made up of logically contiguous data blocks. Data blocks can be physically spread out on disk because of RAID striping and file system implementations.

- Allocation of Extents
  By default, the database allocates an initial extent for a data segment when the segment is created. An extent is always contained in one data file.
- Deallocation of Extents
  In general, the extents of a user segment do not return to the tablespace unless you drop the object using a `DROP` statement.

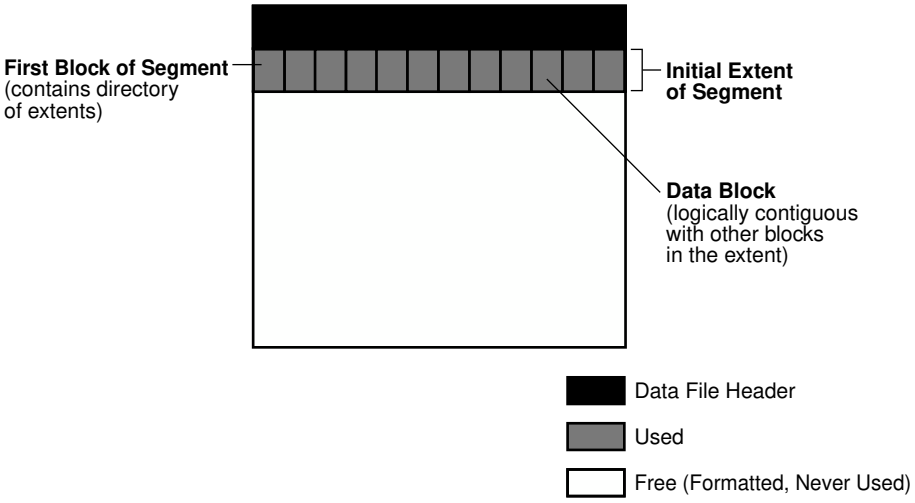
- Storage Parameters for Extents
  Every segment is defined by storage parameters expressed in terms of extents. These parameters control how Oracle Database allocates free space for a segment.

## Allocation of Extents

By default, the database allocates an initial extent for a data segment when the segment is created. An extent is always contained in one data file.

Although no data has been added to the segment, data blocks in the initial extent are reserved for this segment exclusively. The first data block of every segment contains a directory of the extents in the segment. Figure 14-17 shows the initial extent in a segment in a data file that previously contained no data.

**Figure 14-17 Initial Extent of a Segment**

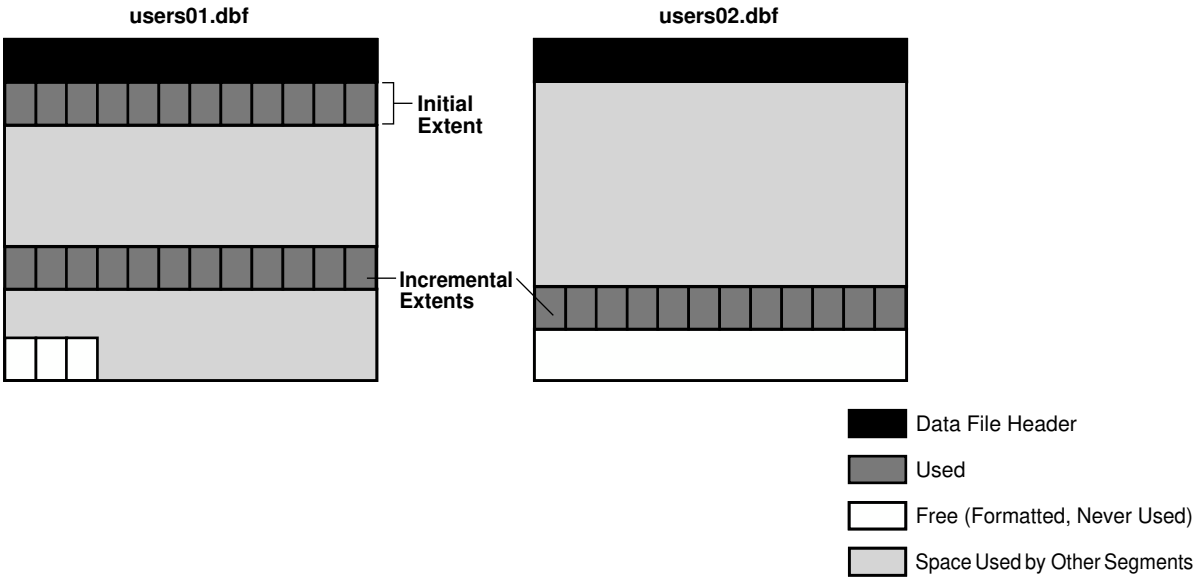



If the initial extent become full, and if more space is required, then the database automatically allocates an incremental extent for this segment. An incremental extent is a subsequent extent created for the segment.

The allocation algorithm depends on whether the tablespace is locally managed or dictionary-managed. In the locally managed case, the database searches the bitmap of a data file for adjacent free blocks. If the data file has insufficient space, then the database looks in another data file. Extents for a segment are always in the same tablespace but may be in different data files.

Figure 14-18 shows that the database can allocate extents for a segment in any data file in the tablespace. For example, the segment can allocate the initial extent in `users01.dbf`, allocate the first incremental extent in `users02.dbf`, and then allocate the next extent in `users01.dbf`.

**Figure 14-18 Incremental Extent of a Segment**




The blocks of a newly allocated extent, although they were free, may not be empty of old data. In ASSM, Oracle Database formats the blocks of a newly allocated extent when it starts using the extent, but only as needed.

> **Note:**
>
> This section applies to serial operations, in which one server process parses and runs a statement. The database allocates extents differently in parallel SQL statements, which entail multiple server processes.

> **See Also:**
>
> - "Segment Space and the High Water Mark"
> - *Oracle Database Administrator's Guide* to learn how to manually allocate extents

## Deallocation of Extents

In general, the extents of a user segment do not return to the tablespace unless you drop the object using a `DROP` statement.

For example, if you delete all rows in a table, then the database does not reclaim the data blocks for use by other objects in the tablespace. You can also drop the segment using the `DBMS_SPACE_ADMIN` package.

> **Note:**
>
> In an undo segment, Oracle Database periodically deallocates one or more extents if it has the `OPTIMAL` size specified or if the database is in automatic undo management mode.

In some circumstances, you can manually deallocate space. The Oracle Segment Advisor helps determine whether an object has space available for reclamation based on the level of fragmentation in the object. The following techniques can free extents:

- Use an online segment shrink to reclaim fragmented space in a segment. Segment shrink is an online, in-place operation. In general, data compaction leads to better cache utilization and requires fewer blocks to be read in a full table scan.
- Move the data of a nonpartitioned table or table partition into a new segment, and optionally into a different tablespace for which you have quota.
- Rebuild or coalesce the index.
- Truncate a table or table cluster, which removes all rows. By default, Oracle Database deallocates all space used by the removed rows except that specified by the `MINEXTENTS` storage parameter. Starting in Oracle Database 11g Release 2 (11.2.0.2), you can also use `TRUNCATE` with the `DROP ALL STORAGE` option to drop entire segments.
- Deallocate unused space, which frees the unused space at the high water mark end of the database segment and makes the space available for other segments in the tablespace.

When extents are freed, Oracle Database modifies the bitmap in the data file for locally managed tablespaces to reflect the regained extents as available space. Any data in the blocks of freed extents becomes inaccessible.

> **See Also:**
>
> - "Coalescing an Index Block"
> - "Undo Tablespaces"
> - "Segment Space and the High Water Mark"
> - *Oracle Database Administrator's Guide* to learn how to reclaim segment space

## Storage Parameters for Extents

Every segment is defined by storage parameters expressed in terms of extents. These parameters control how Oracle Database allocates free space for a segment.

The storage settings are determined in the following order of precedence, with settings higher on the list overriding settings lower on the list:

1. Segment storage clause
2. Tablespace storage clause
3. Oracle Database default

A locally managed tablespace can have either uniform extent sizes or variable extent sizes determined automatically by the system:

- For uniform extents, you can specify an extent size or use the default size of 1 MB. All extents in the tablespace are of this size. Locally managed temporary tablespaces can only use this type of allocation.
- For automatically allocated extents, Oracle Database determines the optimal size of additional extents.

For locally managed tablespaces, some storage parameters cannot be specified at the tablespace level. However, you can specify these parameters at the segment level. In this case, the database uses all parameters together to compute the initial size of the segment. Internal algorithms determine the subsequent size of each extent.

**See Also:**

- *Oracle Database Administrator's Guide* to learn about extent management considerations when creating a locally managed tablespace
- *Oracle Database SQL Language Reference* to learn about options in the storage clause

# Overview of Segments

A segment is a set of extents that contains all the data for a logical storage structure within a tablespace.

For example, Oracle Database allocates one or more extents to form the data segment for a table. The database also allocates one or more extents to form the index segment for an index on a table.

Oracle Database manages segment space automatically or manually. This section assumes the use of ASSM.

- User Segments
  A single data segment in a database stores the data for one user object.
- Temporary Segments
  When processing a query, Oracle Database often requires temporary workspace for intermediate stages of SQL statement execution.
- Undo Segments
  Oracle Database maintains records of the actions of transactions, collectively known as **undo data**.
- Segment Space and the High Water Mark
  To manage space, Oracle Database tracks the state of blocks in the segment. The **high water mark (HWM)** is the point in a segment beyond which data blocks are unformatted and have never been used.

> **See Also:**
>
> "Logical Space Management" to learn more about ASSM

# User Segments

A single data segment in a database stores the data for one user object.

There are different types of segments. Examples of user segments include:

- Table, table partition, or table cluster
- LOB or LOB partition
- Index or index partition

Each nonpartitioned object and object partition is stored in its own segment. For example, if an index has five partitions, then five segments contain the index data.

- User Segment Creation
  By default, the database uses deferred segment creation to update only database metadata when creating tables, indexes, and partitions.

## User Segment Creation

By default, the database uses deferred segment creation to update only database metadata when creating tables, indexes, and partitions.

When a user inserts the first row into a table or partition, the database creates segments for the table or partition, its LOB columns, and its indexes. Deferred segment creation avoids using database resources unnecessarily. For example, installation of an application can create thousands of objects, consuming significant disk space. Many of these objects may never be used.

The `DBMS_SPACE_ADMIN` package manages segments for empty objects. You can use this PL/SQL package to do the following:

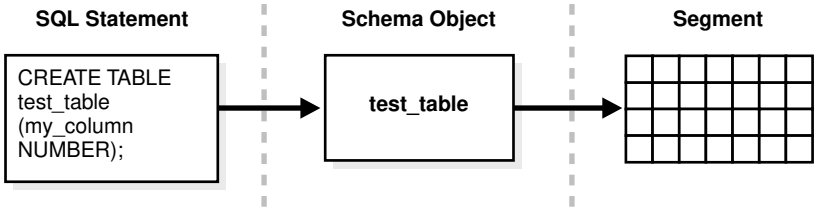
- Manually materialize segments for empty tables or partitions that do not have segments created
- Remove segments from empty tables or partitions that currently have an empty segment allocated

To best illustrate the relationship between object creation and segment creation, assume that deferred segment creation is disabled. You create a table as follows:

```
CREATE TABLE test_table (my_column NUMBER);
```

As shown in Figure 14-19, the database creates one segment for the table.

**Figure 14-19 Creation of a User Segment**

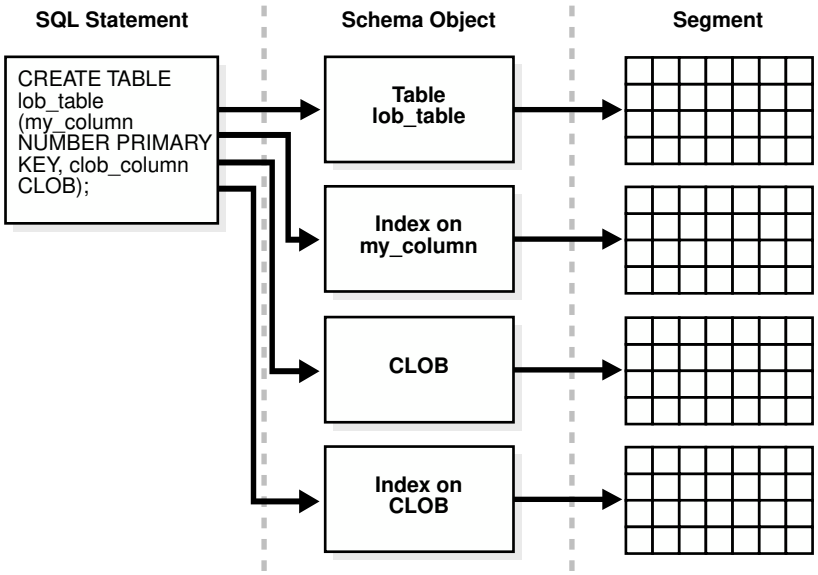



When you create a table with a primary key or unique key, Oracle Database automatically creates an index for this key. Again assume that deferred segment creation is disabled. You create a table as follows:

```
CREATE TABLE lob_table (my_column NUMBER PRIMARY KEY, clob_column CLOB);
```

Figure 14-20 shows that the data for `lob_table` is stored in one segment, while the implicitly created index is in a different segment. Also, the CLOB data is stored in its own segment, as is its associated CLOB index. Thus, the `CREATE TABLE` statement results in the creation of *four* different segments.

**Figure 14-20 Multiple Segments**




> **Note:**
>
> The segments of a table and the index for this table do not have to occupy the same tablespace.

The database allocates one or more extents when a segment is created. Storage parameters for the object determine how the extents for each segment are allocated. The parameters

affect the efficiency of data retrieval and storage for the data segment associated with the object.

**See Also:**

- *Oracle Database SecureFiles and Large Objects Developer's Guide* to learn about internal LOBs
- "Storage Parameters for Extents "
- *Oracle Database Administrator's Guide* to learn how to manage deferred segment creation
- *Oracle Database SQL Language Reference* for `CREATE TABLE` syntax

## Temporary Segments

When processing a query, Oracle Database often requires temporary workspace for intermediate stages of SQL statement execution.

Typical operations that may require a temporary segment include sorting, hashing, and merging bitmaps. While creating an index, Oracle Database also places index segments into temporary segments and then converts them into permanent segments when the index is complete.

Oracle Database does not create a temporary segment if an operation can be performed in memory. However, if memory use is not possible, then the database automatically allocates a temporary segment on disk.

- Allocation of Temporary Segments for Queries
  Oracle Database allocates temporary segments for queries as needed during a user session and drops them when the query completes. Changes to temporary segments are not recorded in the online redo log, except for space management operations on the temporary segment.
- Allocation of Segments for Temporary Tables and Indexes
  Oracle Database can allocate temporary segments for temporary tables and their indexes.

## Allocation of Temporary Segments for Queries

Oracle Database allocates temporary segments for queries as needed during a user session and drops them when the query completes. Changes to temporary segments are not recorded in the online redo log, except for space management operations on the temporary segment.

The database creates temporary segments in the temporary tablespace assigned to the user. The default storage characteristics of the tablespace determine the characteristics of the extents in the temporary segment. Because allocation and deallocation of temporary segments occurs frequently, the best practice is to create at least one special tablespace for temporary segments. The database distributes I/O across disks and avoids fragmenting `SYSTEM` and other tablespaces with temporary segments.

> **Note:**
>
> When `SYSTEM` is locally managed, you must define a default temporary tablespace at database creation. A locally managed `SYSTEM` tablespace cannot be used for default temporary storage.

> **See Also:**
>
> - "Overview of the Online Redo Log"
> - *Oracle Database Administrator's Guide* to learn how to create temporary tablespaces
> - *Oracle Database SQL Language Reference* for `CREATE TEMPORARY TABLESPACE` syntax and semantics

## Allocation of Segments for Temporary Tables and Indexes

Oracle Database can allocate temporary segments for temporary tables and their indexes.

Temporary tables hold data that exists only for the duration of a transaction or session. Each session accesses only the extents allocated for itself and cannot access extents allocated for other sessions.

Oracle Database allocates segments for a global temporary table when the first `INSERT` into the table occurs, and for a private temporary table only when needed. The insertion can occur explicitly or because of `CREATE TABLE AS SELECT`. The database allocates the segments for the table and its indexes, creates the root page for the indexes, and allocates any `LOB` segments.

A temporary tablespace of the current user allocates segments for a temporary table. For example, the temporary tablespace assigned to `user1` is `temp1` and the temporary tablespace assigned to `user2` is `temp2`. In this case, `user1` stores temporary data in the `temp1` segments, while `user2` stores temporary data in the `temp2` segments.

> **See Also:**
>
> - "Overview of Temporary Tables"
> - *Oracle Database Administrator's Guide* to learn how to create temporary tables

# Undo Segments

Oracle Database maintains records of the actions of transactions, collectively known as **undo data**.

Oracle Database uses undo data to do the following:

- Roll back an active transaction

- Recover a terminated transaction
- Provide read consistency
- Perform some logical flashback operations

Oracle Database stores undo data inside the database rather than in external logs. Undo data is stored in blocks that are updated just like data blocks, with changes to these blocks generating redo records. In this way, Oracle Database can efficiently access undo data without needing to read external logs.

Undo data for permanent objects is stored in an undo tablespace. Oracle Database provides a fully automated mechanism, known as automatic undo management mode, for managing undo segments and space in an undo tablespace.

The database separates undo data into two streams. A temporary undo stream encapsulates only undo records generated by changes to temporary objects, whereas a permanent undo stream encapsulates only undo records for permanent objects. The database manages temporary and permanent undo independently. Undo separation decreases storage and enhances performance by doing the following:

- Enabling you to configure permanent and undo tablespace sizes that best fit the workloads for permanent and temporary tables
- Reducing the size of redo written to the online redo log
- Avoiding the need to back up temporary undo data

On an Active Data Guard instance, DML on global temporary tables requires undo to be generated in temporary undo segments.

- Undo Segments and Transactions
  When a transaction starts, the database binds (assigns) the transaction to an undo segment, and therefore to a **transaction table**, in the current undo tablespace. In rare circumstances, if the database instance does not have a designated undo tablespace, then the transaction binds to the system undo segment.
- Transaction Rollback
  When a `ROLLBACK` statement is issued, the database uses undo records to roll back changes made to the database by the uncommitted transaction.
- Temporary Undo Segments
  A **temporary undo segment** is an optional space management container for temporary undo data only.

**See Also:**

- "Use of the Online Redo Log"
- "Temporary Undo Segments"
- *Oracle Database Administrator's Guide* to learn about temporary undo segments
- *Oracle Database Reference* to learn about the `TEMP_UNDO_ENABLED` initialization parameter

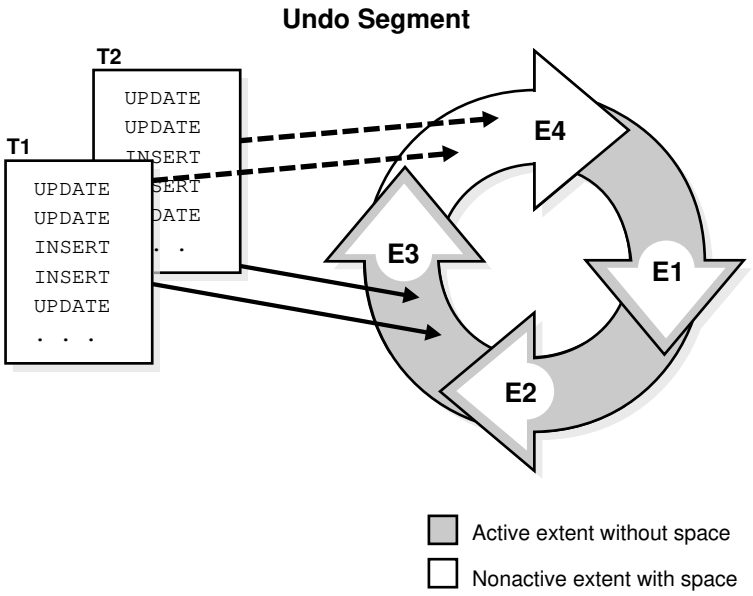
## Undo Segments and Transactions

When a transaction starts, the database binds (assigns) the transaction to an undo segment, and therefore to a **transaction table**, in the current undo tablespace. In rare circumstances, if the database instance does not have a designated undo tablespace, then the transaction binds to the system undo segment.

Multiple active transactions can write concurrently to the same undo segment or to different segments. For example, transactions T1 and T2 can both write to undo segment U1, or T1 can write to U1 while T2 writes to undo segment U2.

Conceptually, the extents in an undo segment form a ring. Transactions write to one undo extent, and then to the next extent in the ring, and so on in cyclical fashion. Figure 14-21 shows two transactions, T1 and T2, which begin writing in the third extent (E3) of an undo segment and continue writing to the fourth extent (E4).

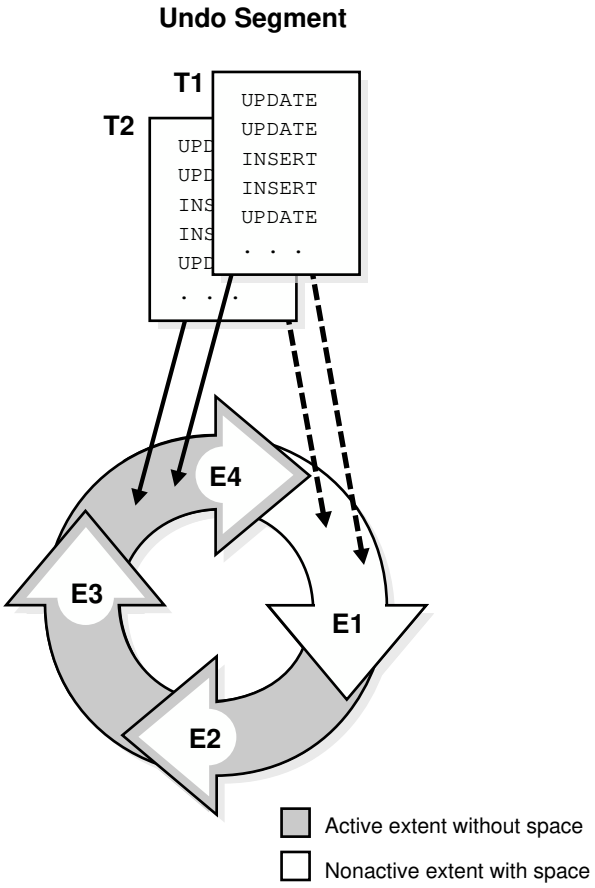
**Figure 14-21 Ring of Allocated Extents in an Undo Segment**




At any given time, a transaction writes sequentially to only one extent in an undo segment, known as the *current extent* for the transaction. Multiple active transactions can write simultaneously to the same current extent or to different current extents. Figure 14-21 shows transactions T1 and T2 writing simultaneously to extent E3. Within an undo extent, a data block contains data for only one transaction.

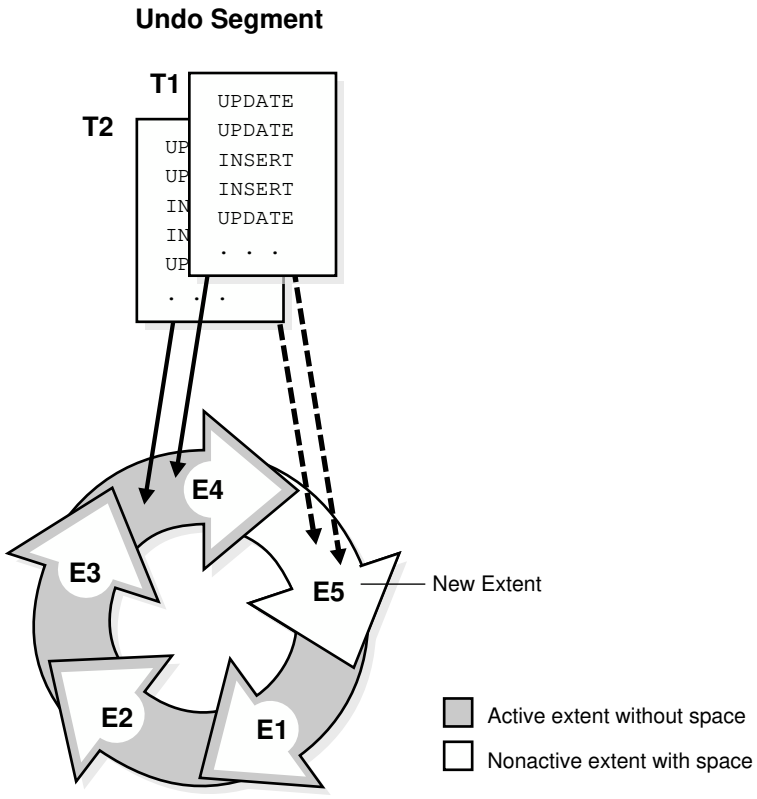
As the current undo extent fills, the first transaction needing space checks the availability of the next allocated extent in the ring. If the next extent does *not* contain data from an active transaction, then this extent becomes the current extent. Now all transactions that need space can write to the new current extent. In Figure 14-22, when E4 is full, T1 and T2 continue writing to E1, overwriting the nonactive undo data in E1.

**Figure 14-22 Cyclical Use of Allocated Extents in an Undo Segment**




If the next extent *does* contain data from an active transaction, then the database must allocate a new extent. Figure 14-23 shows a scenario in which T1 and T2 are writing to E4. When E4 fills up, the transactions cannot continue writing to E1 because E1 contains active undo entries. Therefore, the database allocates a new extent (E5) for this undo segment. The transactions continue writing to E5.

**Figure 14-23 Allocation of a New Extent for an Undo Segment**




> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage undo segments

## Transaction Rollback

When a `ROLLBACK` statement is issued, the database uses undo records to roll back changes made to the database by the uncommitted transaction.

During recovery, the database rolls back any uncommitted changes applied from the online redo log to the data files. Undo records provide read consistency by maintaining the before image of the data for users accessing data at the same time that another user is changing it.

## Temporary Undo Segments

A **temporary undo segment** is an optional space management container for temporary undo data only.

Undo records for changes to temporary tables are both session-specific and useful only for read consistency and transaction rollback. Before Oracle Database 12c, the database always stored these records in the online redo log. Because changes to temporary objects are not logged in the online redo log, writing undo for temporary objects into temporary undo segments

saves space in the online redo log and archived redo log files. The database does not log changes to the undo or changes to the temporary table, which improves performance.

You can set the `TEMP_UNDO_ENABLED` initialization parameter so that temporary tables store undo data in a temporary undo segment. When this parameter is `TRUE`, the database allocates temporary undo segments from temporary tablespaces.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about temporary undo segments
> - *Oracle Database Reference* to learn about the `TEMP_UNDO_ENABLED` initialization parameter

## Segment Space and the High Water Mark

To manage space, Oracle Database tracks the state of blocks in the segment. The **high water mark (HWM)** is the point in a segment beyond which data blocks are unformatted and have never been used.

MSSM uses free lists to manage segment space. At table creation, no blocks in the segment are formatted. When a session first inserts rows into the table, the database searches the free list for usable blocks. If the database finds no usable blocks, then it preformats a group of blocks, places them on the free list, and begins inserting data into the blocks. In MSSM, a full table scan reads *all* blocks below the HWM.

ASSM does not use free lists and so must manage space differently. When a session first inserts data into a table, the database formats a single bitmap block instead of preformatting a group of blocks as in MSSM. The bitmap tracks the state of blocks in the segment, taking the place of the free list. The database uses the bitmap to find free blocks and then formats each block before filling it with data. ASSM spread out inserts among blocks to avoid concurrency issues.

Every data block in an ASSM segment is in one of the following states:

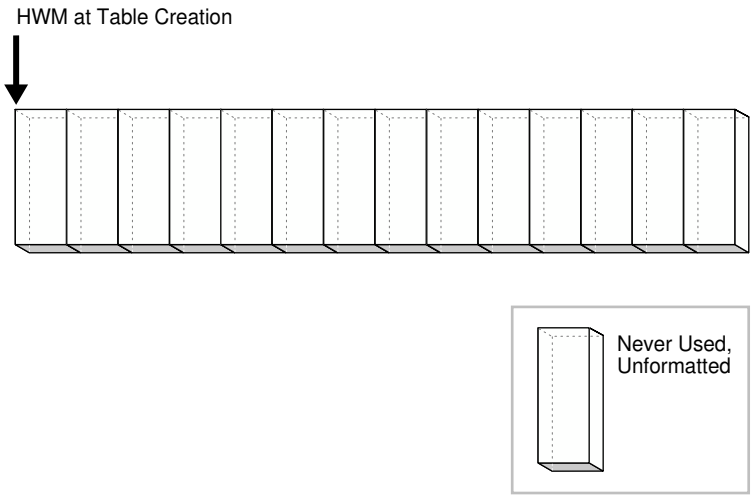
- Above the HWM

  These blocks are unformatted and have never been used.
- Below the HWM

  These blocks are in one of the following states:
  - Allocated, but currently unformatted and unused
  - Formatted and contain data
  - Formatted and empty because the data was deleted

Figure 14-24 depicts an ASSM segment as a horizontal series of blocks. At table creation, the HWM is at the beginning of the segment on the left. Because no data has been inserted yet, all blocks in the segment are unformatted and never used.

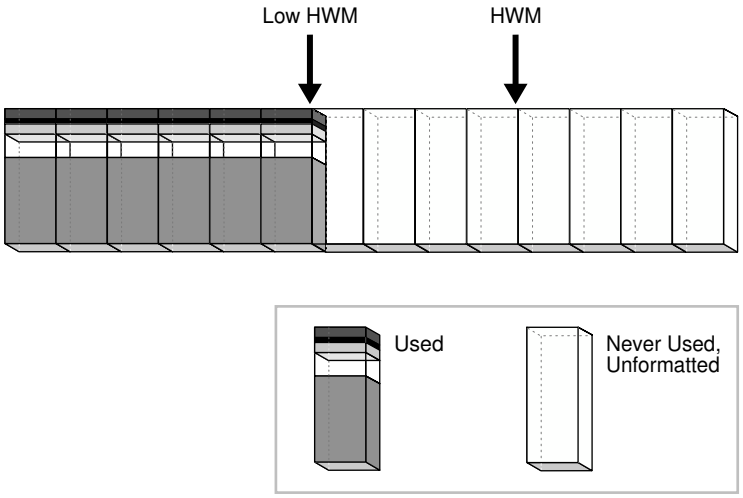
**Figure 14-24 HWM at Table Creation**




Suppose that a transaction inserts rows into the segment. The database must allocate a group of blocks to hold the rows. The allocated blocks fall below the HWM. The database formats a bitmap block in this group to hold the metadata, but does not preformat the remaining blocks in the group.

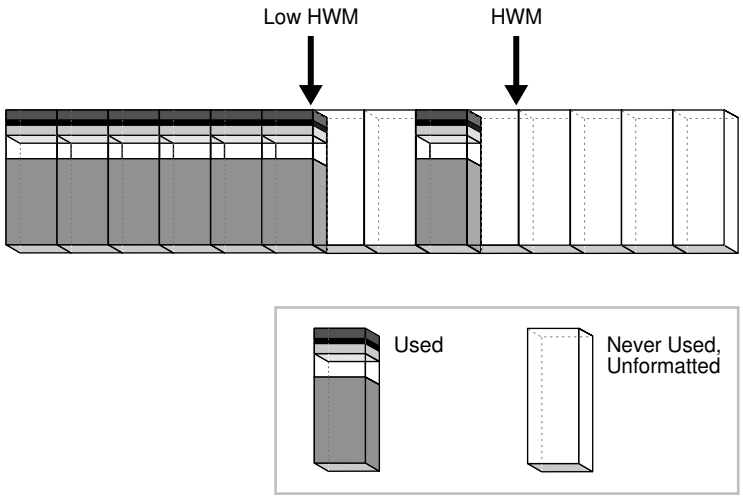
In Figure 14-25, the blocks below the HWM are allocated, whereas blocks above the HWM are neither allocated or formatted. As inserts occur, the database can write to any block with available space. The low high water mark (low HWM) marks the point below which all blocks are known to be formatted because they either contain data or formerly contained data.

**Figure 14-25 HWM and Low HWM**




In Figure 14-26, the database chooses a block between the HWM and low HWM and writes to it. The database could have just as easily chosen any other block between the HWM and low HWM, or any block below the low HWM that had available space. In Figure 14-26, the blocks to either side of the newly filled block are unformatted.

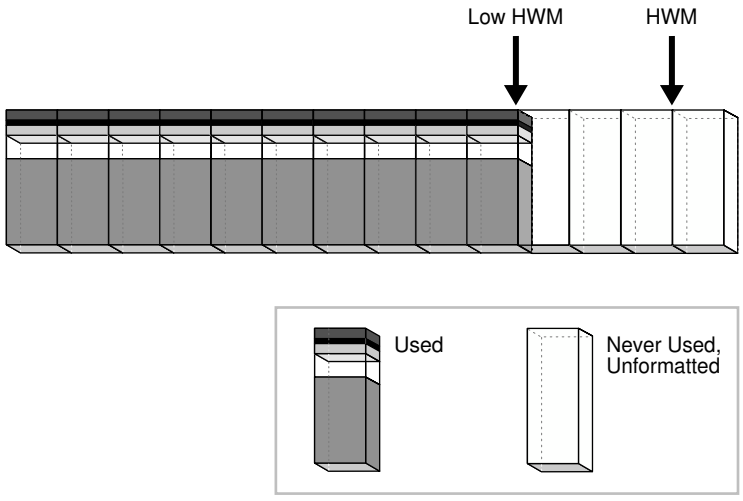
**Figure 14-26 HWM and Low HWM**




The low HWM is important in a full table scan. Because blocks below the HWM are formatted only when used, some blocks could be unformatted, as in Figure 14-26. For this reason, the database reads the bitmap block to obtain the location of the low HWM. The database reads all blocks up to the low HWM because they are known to be formatted, and then carefully reads only the formatted blocks between the low HWM and the HWM.

Assume that a new transaction inserts rows into the table, but the bitmap indicates that insufficient free space exists under the HWM. In Figure 14-27, the database advances the HWM to the right, allocating a new group of unformatted blocks.

**Figure 14-27 Advancing HWM and Low HWM**




When the blocks between the HWM and low HWM are full, the HWM advances to the right and the low HWM advances to the location of the old HWM. As the database inserts data over time, the HWM continues to advance to the right, with the low HWM always trailing behind it. Unless you manually rebuild, truncate, or shrink the object, the HWM never retreats.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to shrink segments online
- *Oracle Database SQL Language Reference* for `TRUNCATE TABLE` syntax and semantics

# Overview of Tablespaces

A **tablespace** is a logical storage container for segments.

Segments are database objects, such as tables and indexes, that consume storage space. At the physical level, a tablespace stores data in one or more data files or temp files.

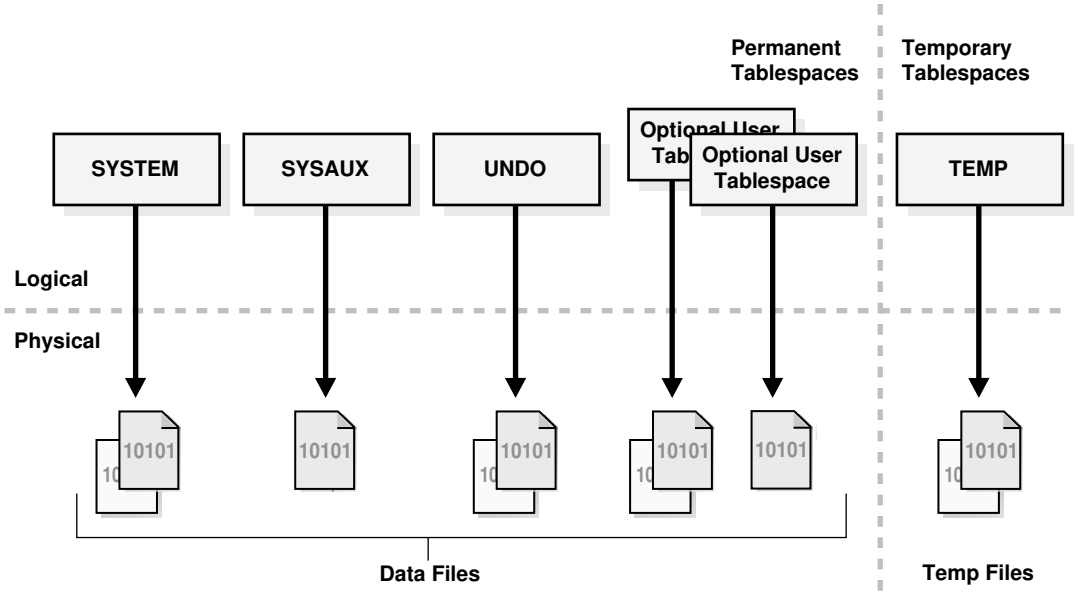
- Tablespaces in a Multitenant Environment
  In a CDB, each PDB and application root has its own set of tablespaces.
- Permanent Tablespaces
  A **permanent tablespace** groups persistent schema objects. The segments for objects in the tablespace are stored physically in data files.
- Temporary Tablespaces
  A **temporary tablespace** contains transient data that persists only for the duration of a session. No permanent schema objects can reside in a temporary tablespace. A **temp file** stores temporary tablespace data.
- Tablespace Modes
  The tablespace mode determines the accessibility of the tablespace.
- Tablespace File Size
  A tablespace is either a **bigfile tablespace** or a **smallfile tablespace**. These tablespaces are indistinguishable in terms of execution of SQL statements that do not explicitly refer to data files or temp files.

## Tablespaces in a Multitenant Environment

In a CDB, each PDB and application root has its own set of tablespaces.

Every CDB root, PDB, and application root must have the `SYSTEM` and `SYSAUX` tablespaces. The following figure shows the tablespaces in a typical container.

**Figure 14-28 Tablespaces**




A CDB contains the following:

- One control file
- One online redo log
- One or more undo tablespaces

  Only a common user who has the appropriate privileges and whose current container is the CDB root can create an undo tablespace. At any given time, a CDB is either in either of the following undo modes:

  – Local undo mode

    In this case, each PDB has its own undo tablespace. If a CDB is using local undo mode, then the database automatically creates an undo tablespace in every PDB. Local undo provides advantages such as the ability to perform a hot clone of a PDB, and relocate PDBs quickly. Also, local undo provides level of isolation and enables faster unplug and point-in-time recovery operations.

    A local undo tablespace is required for each node in an Oracle Real Application Clusters (RAC) cluster in which the PDB is open. For example, if you move a PDB from a two-node cluster to a four-node cluster, and if the PDB is open in all nodes, then the database automatically creates the additional required undo tablespaces. If you move the PDB back again, then you can drop the redundant undo tablespaces.

    > **Note:**
    >
    > By default, Database Configuration Assistant (DBCA) creates new CDBs with local undo enabled.

  – Shared undo mode

In a single-instance CDB, only one active undo tablespace exists. For an Oracle RAC CDB, one active undo tablespace exists for every instance. All undo tablespaces are visible in the data dictionaries and related views of all containers.

The undo mode applies to the entire CDB, which means that either every container uses shared undo, or every container uses local undo. You can switch between undo modes in a CDB, which necessitates re-starting the database.

- `SYSTEM` and `SYSAUX` tablespaces for every container

  The CDB root, each application root, and each PDB has its own `SYSTEM` and `SYSAUX` tablespaces. Each container also has its own set of data dictionary tables describing the objects that reside in the container.

- Zero or more user-created tablespaces

  In a typical use case, each PDB has its own set of non-system tablespaces. These tablespaces contain the data for user-defined schemas and objects in the PDB. You can limit the amount of storage used by the data files for a PDB by using the `STORAGE` clause in a `CREATE PLUGGABLE DATABASE` or `ALTER PLUGGABLE DATABASE` statement.

  The storage of the data dictionary within the PDB enables it to be portable. You can unplug a PDB from a CDB, and plug it in to a different CDB.

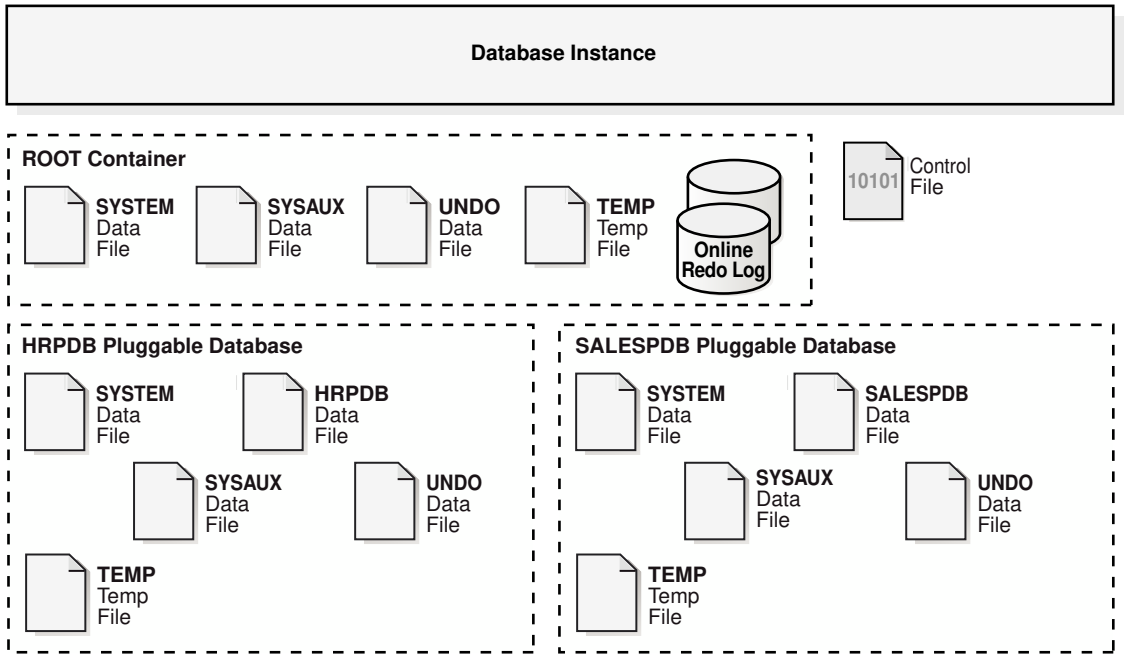
- A set of temp files for every container

  One default temporary tablespace exists for the CDB root, and one for each application root, application PDB, and PDB.

**Example 14-3 CDB in Local Undo Mode**

This example shows aspects of the physical storage architecture of a CDB with two PDBs: `hrpdb` and `salespdb`. In this example, the database uses local undo mode, and so has undo data files in the CDB root, `hrpdb`, and `salespdb`.

**Figure 14-29 Physical Architecture of a CDB in Local Undo Mode**



> **See Also:**
>
> "Data Dictionary Separation in a CDB"

## Permanent Tablespaces

A **permanent tablespace** groups persistent schema objects. The segments for objects in the tablespace are stored physically in data files.

Each database user is assigned a default permanent tablespace. A very small database may need only the default `SYSTEM` and `SYSAUX` tablespaces. However, Oracle recommends that you create at least one tablespace to store user and application data. You can use tablespaces to achieve the following goals:

- Control disk space allocation for database data
- Assign a quota (space allowance or limit) to a database user
- Take individual tablespaces online or offline without affecting the availability of the whole database
- Perform backup and recovery of individual tablespaces
- Import or export application data by using the Oracle Data Pump utility
- Create a transportable tablespace that you can copy or move from one database to another, even across platforms

  Moving data by transporting tablespaces can be orders of magnitude faster than either export/import or unload/load of the same data, because transporting a tablespace involves only copying data files and integrating the tablespace metadata. When you transport tablespaces you can also move index data.

- The SYSTEM Tablespace
  The `SYSTEM` tablespace is a necessary administrative tablespace included with the database when it is created. Oracle Database uses `SYSTEM` to manage the database.
- The SYSAUX Tablespace
  The `SYSAUX` tablespace is an auxiliary tablespace to the `SYSTEM` tablespace.
- Undo Tablespaces
  An **undo tablespace** is a locally managed tablespace reserved for system-managed undo data.
- Shadow Tablespaces
  A **shadow tablespace** is a bigfile tablespace intended for **shadow lost write protection**.

> **See Also:**
>
> - *Oracle Database Utilities*
> - *Oracle Database Administrator's Guide* to learn how to transport tablespaces
> - *Oracle Database Utilities* to learn about Oracle Data Pump

## The SYSTEM Tablespace

The SYSTEM tablespace is a necessary administrative tablespace included with the database when it is created. Oracle Database uses SYSTEM to manage the database.

The SYSTEM tablespace includes the following information, all owned by the SYS user:

- The data dictionary
- Tables and views that contain administrative information about the database
- Compiled stored objects such as triggers, procedures, and packages

The SYSTEM tablespace is managed as any other tablespace, but requires a higher level of privilege and is restricted in some ways. For example, you cannot rename or drop the SYSTEM tablespace.

By default, Oracle Database sets all newly created user tablespaces to be locally managed. In a database with a locally managed SYSTEM tablespace, you cannot create dictionary-managed tablespaces (which are deprecated). However, if you execute the CREATE DATABASE statement manually and accept the defaults, then the SYSTEM tablespace is dictionary managed. You can migrate an existing dictionary-managed SYSTEM tablespace to a locally managed format.

> **Note:**
>
> Oracle strongly recommends that you use Database Configuration Assistant (DBCA) to create new databases so that all tablespaces, including SYSTEM, are locally managed by default.

> **See Also:**
>
> - "Online and Offline Tablespaces" for information about the permanent online condition of the SYSTEM tablespace
> - *Oracle Database Administrator's Guide* to learn how to create or migrate to a locally managed SYSTEM tablespace
> - *Oracle Database SQL Language Reference* for CREATE DATABASE syntax and semantics

## The SYSAUX Tablespace

The SYSAUX tablespace is an auxiliary tablespace to the SYSTEM tablespace.

Because SYSAUX is the default tablespace for many Oracle Database features and products that previously required their own tablespaces, it reduces the number of tablespaces required by the database. It also reduces the load on the SYSTEM tablespace.

Database creation or upgrade automatically creates the SYSAUX tablespace. During normal database operation, the database does not allow the SYSAUX tablespace to be dropped or renamed. If the SYSAUX tablespace becomes unavailable, then core database functionality

remains operational. The database features that use the SYSAUX tablespace could fail, or function with limited capability.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn about the SYSAUX tablespace

## Undo Tablespaces

An **undo tablespace** is a locally managed tablespace reserved for system-managed undo data.

Like other permanent tablespaces, undo tablespaces contain data files. Undo blocks in these files are grouped in extents.

- Automatic Undo Management Mode
  Undo tablespaces require the database to be in the **default automatic undo mode**.
- Automatic Undo Retention
  The **undo retention period** is the minimum amount of time that Oracle Database attempts to retain old undo data before overwriting it.

> **See Also:**
>
> "Undo Segments"

### Automatic Undo Management Mode

Undo tablespaces require the database to be in the **default automatic undo mode**.

Automatic mode eliminates the complexities of manually administering undo segments. The database automatically tunes itself to provide the best possible retention of undo data to satisfy long-running queries that may require this data.

A new installation of Oracle Database automatically creates an undo tablespace. Earlier versions of Oracle Database may not include an undo tablespace and use legacy rollback segments instead, known as manual undo management mode. Oracle Database contains an Undo Advisor that provides advice on and helps automate your undo environment.

A database can contain multiple undo tablespaces, but only one can be in use at a time. When an instance attempts to open a database, Oracle Database automatically selects the first available undo tablespace. If no undo tablespace is available, then the instance starts without an undo tablespace and stores undo data in the SYSTEM tablespace. Storing undo data in SYSTEM is not recommended.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about automatic undo management
> - *Oracle Database Upgrade Guide* to learn how to migrate to automatic undo management mode

### Automatic Undo Retention

The **undo retention period** is the minimum amount of time that Oracle Database attempts to retain old undo data before overwriting it.

Undo retention is important because long-running queries may require older block images to supply read consistency. Also, some Oracle Flashback features can depend on undo availability.

In general, it is desirable to retain old undo data as long as possible. After a transaction commits, undo data is no longer needed for rollback or transaction recovery. The database can retain old undo data if the undo tablespace has space for new transactions. When available space is low, the database begins to overwrite old undo data for committed transactions.

Oracle Database automatically provides the best possible undo retention for the current undo tablespace. The database collects usage statistics and tunes the retention period based on these statistics and the undo tablespace size. If the undo tablespace is configured with the `AUTOEXTEND` option, and if the maximum size is not specified, then undo retention tuning is different. In this case, the database tunes the undo retention period to be slightly longer than the longest-running query, if space allows.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more details on automatic tuning of undo retention

## Shadow Tablespaces

A **shadow tablespace** is a bigfile tablespace intended for **shadow lost write protection**.

> **Note:**
>
> Shadow lost write protection is not related to lost write protection that is configured with the `DB_LOST_WRITE_PROTECT` initialization parameter and a standby database.

- Purpose of Shadow Tablespaces
  Shadow lost write protection provides fast detection and immediate response to a **lost write**.

- How Shadow Tablespaces Work
  Lost write protection requires two tablespaces: a shadow tablespace, and a non-shadow tablespace whose blocks are tracked by the shadow tablespace.
- User Interface for Shadow Tablespaces
  You enable and disable shadow lost write protection using the `ALTER PLUGGABLE DATABASE` command.
- Example: Configuring Lost Write Protection
  This example enables shadow lost write tracking for a set of tablespaces.

## Purpose of Shadow Tablespaces

Shadow lost write protection provides fast detection and immediate response to a **lost write**.

A data block lost write occurs when an I/O subsystem acknowledges the completion of the block write even though the write did not occur or when a former image of the block overwrites the current image.

An undetected lost write can result in data corruption because the incorrect data can be used for other DML transactions. For example, a transaction can read old and incorrect data from one table, and then update hundreds of other tables based on this data. In this way, data corruption can spread throughout the database.

Shadow lost write protection provides the following benefits:

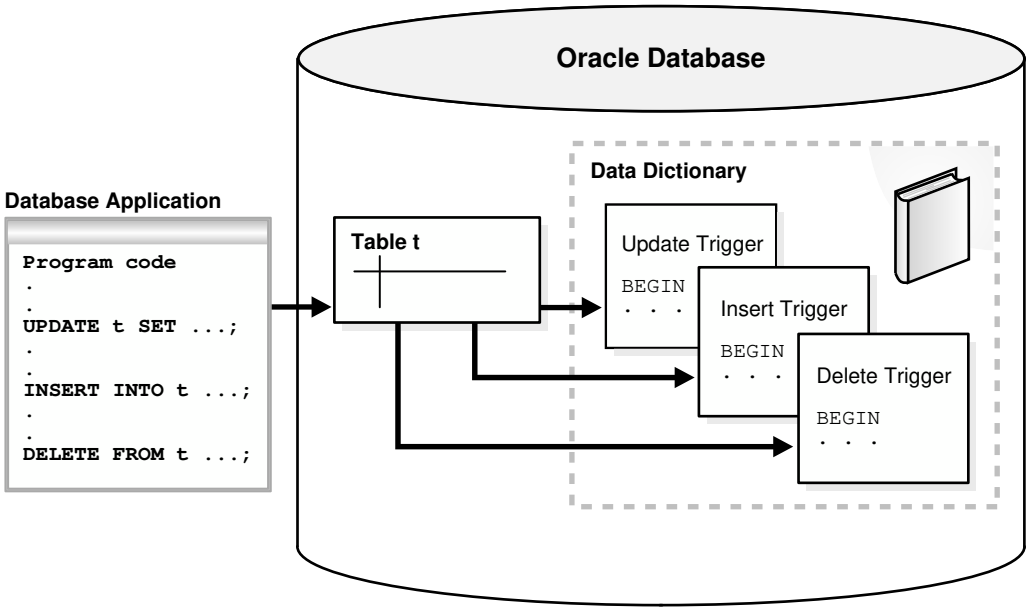
- It detects a lost write before it is consumed for standard DML, SQL*Loader conventional path load, direct path load, and RMAN backups.
- You can enable shadow lost write protection for specific tablespaces and data files. You do not need to track all data.
- You can replace one shadow tablespace with another to change its configuration or location.
- You can suspend and resume shadow lost write protection for a tablespace or data file.
- You can enable or disable it for a PDB with a single `ALTER DATABASE ... LOST WRITE PROTECTION` statement. The `PROP$` table indicates whether tracking is enabled for a PDB.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `LOST WRITE PROTECTION` clause

## How Shadow Tablespaces Work

Lost write protection requires two tablespaces: a shadow tablespace, and a non-shadow tablespace whose blocks are tracked by the shadow tablespace.

The following figure provides a sample scenario. The data files in tablespaces `TBS1` and `TBS2` are tracked by a shadow tablespace. Only data file `DBF6` in tablespace `TBS3` is tracked by the shadow tablespace.




One tracked data file maps to one shadow extent in a shadow tablespace. Every data block in a tracked data file has a corresponding entry in a shadow block. This entry contains the SCN of the tracked data block. When a tracked data block is read from disk, shadow lost write protection compares the SCN for the block in the shadow tablespace with the SCN of the most recent write in the tracked data block. If the shadow entry has an SCN greater than the data block being read, then a lost write has occurred, prompting an error.

The shadow extent is sized with significant extra space to prevent the automatic resizing of data files from causing the shadow extent to grow too large. If a tracked data file is resized either manually or automatically, and if the shadow extent needs to grow, then the database attempts to resize the tracking data. If sufficient space in the shadow tablespaces does not exist, then the database writes a warning to the alert log, and tracks as many data blocks as possible.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage shadow lost write protection

## User Interface for Shadow Tablespaces

You enable and disable shadow lost write protection using the `ALTER PLUGGABLE DATABASE` command.

For shadow lost write protection to protect a specific tablespace or data file, the following conditions must be met:

- You must have enabled shadow lost write protection for the entire PDB by using the `ALTER PLUGGABLE DATABASE ENABLE LOST WRITE PROTECTION` statement.

> **Note:**
>
> In a CDB, if you enable shadow lost write protection in the root, then the PDBs do not inherit it. You must enable shadow lost write protection for every PDB that you want to protect.

- You must have enabled shadow lost write protection for the tablespace or data file to be protected by using the `ENABLE LOST WRITE PROTECTION` clause.

  When you enable shadow lost write protection for a tablespace, all of the data files of the tablespace are protected, and any data files added to the tablespace are also protected. Note that you cannot enable lost write protection on temporary tablespaces or another lost write tablespace.
- You must have created one or more shadow tablespaces by using the `CREATE BIGFILE TABLESPACE` statement with the `LOST WRITE PROTECTION` clause.

Oracle Database assigns a tracked data file to a specific shadow tablespace automatically. You cannot specify which shadow tablespace is used for a particular data file.

The following data dictionary views monitor shadow tablespaces:

- `DBA_TABLESPACES`

  Shows which tablespaces are shadow tablespaces by querying.
- `DBA_DATA_FILES.LOST_WRITE_PROTECT`

  Shows whether lost write protections is enabled for a data file
- `USER_TABLESPACES.LOST_WRITE_PROTECT`

  Shows whether lost write protection is turned on for a specific tablespace. `DBA_DATA_FILES` does not indicate whether lost write is turned on for a tablespace: you must look at `USER_TABLESPACES` instead.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to manage lost write protection with shadow tablespaces
> - *Oracle Database SQL Language Reference* to learn about the `CREATE TABLESPACE` statement
> - *Oracle Database Reference* to learn about `DBA_TABLESPACES`

## Example: Configuring Lost Write Protection

This example enables shadow lost write tracking for a set of tablespaces.

In this example, your goal is to protect the `salestbs` and `hrtbs` tablespaces within a PDB. You also want to protect the `oetbs01.dbf` data file, and only this data file, within the `oetbs` tablespace. You do the following:

1. Log in to the PDB as an administrator.

2. Create a single shadow tablespace as follows:

```
CREATE BIGFILE TABLESPACE shadow_lwp1
  DATAFILE 'shadow_lwp1_df' SIZE 10M LOST WRITE PROTECTION;
```

3. Enable lost write protection for the PDB as follows:

```
ALTER DATABASE ENABLE LOST WRITE PROTECTION;
```

4. Enable shadow lost write protection for the `salestbs` and `hrtbs` tablespaces as follows:

```
ALTER TABLESPACE salestbs ENABLE LOST WRITE PROTECTION;
ALTER TABLESPACE hrtbs ENABLE LOST WRITE PROTECTION;
```

5. Enable shadow lost write protection for the `oetbs01.dbf` data file as follows:

```
ALTER DATABASE DATAFILE 'oetbs01.dbf' ENABLE LOST WRITE PROTECTION;
```

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to manage shadow tablespaces
- *Oracle Database SQL Language Reference* to learn about the `CREATE TABLESPACE` statement

# Temporary Tablespaces

A **temporary tablespace** contains transient data that persists only for the duration of a session. No permanent schema objects can reside in a temporary tablespace. A **temp file** stores temporary tablespace data.

Temporary tablespaces can improve the concurrency of multiple sort operations that do not fit in memory. These tablespaces also improve the efficiency of space management operations during sorts.

- Shared and Local Temporary Tablespaces
  Temporary tablespaces are either shared or local.
- Default Temporary Tablespaces
  Every database user account is assigned a default shared temporary tablespace. If the database contains local temporary tablespaces, then every user account is also assigned default local temporary storage.

## Shared and Local Temporary Tablespaces

Temporary tablespaces are either shared or local.

A shared temporary tablespace stores temp files on shared disk, so that the temporary space is accessible to all database instances. In contrast, a local temporary tablespace stores separate, non-shared temp files for every database instance. Local temporary tablespaces are useful for Oracle Real Application Clusters or Oracle Flex Clusters.

You can create local temporary tablespaces for both read-only and read/write database instances. When many read-only instances access a single database, local temporary tablespaces can improve performance for queries that involve sorts, hash aggregations, and joins. The advantages are:

- Improving I/O performance by using local rather than shared disk storage
- Avoiding expensive cross-instance temporary space management
- Improving instance startup performance by eliminating on-disk space metadata management

The following table compares the characteristics of shared and local temporary tablespaces.

**Table 14-2 Shared and Local Temporary Tablespaces**

| Shared Temporary Tablespace | Local Temporary Tablespace |
|---|---|
| Created with the `CREATE TEMPORARY TABLESPACE` statement. | Created with the `CREATE LOCAL TEMPORARY TABLESPACE` statement.<br>**Note:** A local temporary tablespaces is always a bigfile tablespace, but the `BIGFILE` keyword is not required in the creation statement. |
| Creates a single temporary tablespace for the database. | Creates separate temporary tablespaces for every database instance. The `FOR LEAF` option creates tablespaces only for read-only instances. The `FOR ALL` option creates tablespaces for all instances, both read-only and read/write. |
| Supports tablespace groups. | Does not support tablespace groups. |
| Stores temp file metadata in the control file. | Stores temp file metadata common to all instances in the control file, and instance-specific metadata (for example, the bitmaps for allocation, current temp file sizes, and file status) in the SGA. |

**See Also:**

"Introduction to the Oracle Database Instance"

## Default Temporary Tablespaces

Every database user account is assigned a default shared temporary tablespace. If the database contains local temporary tablespaces, then every user account is also assigned default local temporary storage.

You can specify a different temporary tablespace for a user account with the `CREATE USER` or `ALTER USER` statements. Oracle Database use the system-level default temporary tablespace for users for whom you do not specify a different temporary tablespace.

- Creation of Default Temporary Tablespaces
  When creating a database, the default temporary storage depends on whether the `SYSTEM` tablespace is locally managed.

- Access to Temporary Storage
  If a user has a temporary tablespace assigned, then the database accesses it first; otherwise, the database accesses the default temporary tablespace. After the database accesses a temporary tablespace for a query, it does not switch to a different one.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `CREATE USER` statement

## Creation of Default Temporary Tablespaces

When creating a database, the default temporary storage depends on whether the `SYSTEM` tablespace is locally managed.

The following table shows how Oracle Database chooses default temporary tablespaces at database creation.

**Table 14-3 Creation of Default Temporary Tablespaces**

| Is the SYSTEM tablespace locally managed? | Does the CREATE DATABASE statement specify a default temporary tablespace? | Then the database ... |
|---|---|---|
| Yes | Yes | Uses the specified tablespace as the default. |
| Yes | No | Creates a temporary tablespace. |
| No | Yes | Uses the specified tablespace as the default. |
| No | No | Uses `SYSTEM` for default temporary storage. The database writes a warning in the alert log saying that a default temporary tablespace is recommended. |

After database creation, you can change the default temporary tablespace for the database with the `ALTER DATABASE DEFAULT TEMPORARY TABLESPACE` statement.

> **Note:**
>
> You cannot make a default temporary tablespace permanent.

> **See Also:**
>
> - "Permanent and Temporary Data Files"
> - *Oracle Database Administrator's Guide* to learn how to create a default temporary tablespace
> - *Oracle Database SQL Language Reference* for the syntax of the `DEFAULT TEMPORARY TABLESPACE` clause of `CREATE DATABASE` and `ALTER DATABASE`

### Access to Temporary Storage

If a user has a temporary tablespace assigned, then the database accesses it first; otherwise, the database accesses the default temporary tablespace. After the database accesses a temporary tablespace for a query, it does not switch to a different one.

A user query can access either shared or local temporary storage. Furthermore, a user could have one default local temporary tablespace assigned for read-only instances, and a different default local temporary tablespace assigned for read/write instances.

For read/write instances, the database gives higher priority to shared temporary tablespaces. For read-only instances, the database gives higher priority to local temporary tablespaces. If the database instance is read/write, then the database searches for space in the following order:

1. Is a shared temporary tablespace assigned to the user?
2. Is a local temporary tablespace assigned to the user?
3. Does the database default temporary tablespace have space?

If the answer to any preceding question is yes, then the database stops the search and allocates space from the specified tablespace; otherwise, space is allocated from the database default local temporary tablespace.

If the database instance is read-only, then the database searches for space in the following order:

1. Is a local temporary tablespace assigned to the user?
2. Does the database default local temporary tablespace assigned have space?
3. Is a shared temporary tablespace assigned to the user?

If the answer to any preceding questions is yes, then the database stops the search and allocates space from the specified tablespace; otherwise, space is allocated from the database default shared temporary tablespace.

## Tablespace Modes

The tablespace mode determines the accessibility of the tablespace.

- Read/Write and Read-Only Tablespaces
  Every tablespace is in a write mode that specifies whether it can be written to.
- Online and Offline Tablespaces
  A tablespace can be online (accessible) or offline (not accessible) whenever the database is open.

## Read/Write and Read-Only Tablespaces

Every tablespace is in a write mode that specifies whether it can be written to.

The mutually exclusive modes are as follows:

- Read/write mode

  Users can read and write to the tablespace. All tablespaces are initially created as read/write. The `SYSTEM` and `SYSAUX` tablespaces and temporary tablespaces are permanently read/write, which means that they cannot be made read-only.

- Read-only mode

  Write operations to the data files in the tablespace are prevented. A read-only tablespace can reside on read-only media such as DVDs or WORM drives.

  Read-only tablespaces eliminate the need to perform backup and recovery of large, static portions of a database. Read-only tablespaces do not change and thus do not require repeated backup. If you recover a database after a media failure, then you do not need to recover read-only tablespaces.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to change a tablespace to read only or read/write mode
- *Oracle Database SQL Language Reference* for `ALTER TABLESPACE` syntax and semantics
- *Oracle Database Backup and Recovery User's Guide* for more information about recovery

## Online and Offline Tablespaces

A tablespace can be online (accessible) or offline (not accessible) whenever the database is open.

A tablespace is usually online so that its data is available to users. The `SYSTEM` tablespace and temporary tablespaces cannot be taken offline.

A tablespace can go offline automatically or manually. For example, you can take a tablespace offline for maintenance or backup and recovery. The database automatically takes a tablespace offline when certain errors are encountered, as when the database writer (DBW) process fails in several attempts to write to a data file. Users trying to access tables in an offline tablespace receive an error.

When a tablespace goes offline, the database does the following:

- The database does not permit subsequent DML statements to reference objects in the offline tablespace. An offline tablespace cannot be read or edited by any utility other than Oracle Database.
- Active transactions with completed statements that refer to data in that tablespace are not affected at the transaction level.

- The database saves undo data corresponding to those completed statements in a deferred undo segment in the `SYSTEM` tablespace. When the tablespace is brought online, the database applies the undo data to the tablespace, if needed.

> **See Also:**
>
> - "Online and Offline Data Files"
> - "Database Writer Process (DBW)"
> - *Oracle Database Administrator's Guide* to learn how to alter tablespace availability

## Tablespace File Size

A tablespace is either a **bigfile tablespace** or a **smallfile tablespace**. These tablespaces are indistinguishable in terms of execution of SQL statements that do not explicitly refer to data files or temp files.

The difference is as follows:

- A smallfile tablespace can contain multiple data files or temp files, but the files cannot be as large as in a bigfile tablespace. This is the default tablespace type.
- A bigfile tablespace contains one very large data file or temp file. This type of tablespace can do the following:
  - Increase the storage capacity of a database

    The maximum number of data files in a database is limited, so increasing the size of each data file increases the overall storage.
  - Reduce the burden of managing many data files and temp files

    Bigfile tablespaces simplify file management with Oracle Managed Files and Automatic Storage Management (Oracle ASM) by eliminating the need for adding new files and dealing with multiple files.
  - Perform operations on tablespaces rather than individual files

    Bigfile tablespaces make the tablespace the main unit of the disk space administration, backup and recovery, and so on.

  Bigfile tablespaces are supported only for locally managed tablespaces with ASSM. However, locally managed undo and temporary tablespaces can be bigfile tablespaces even when segments are manually managed.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* to learn more about backup and recovery
> - *Oracle Database Administrator's Guide* to learn how to manage bigfile tablespaces

# Part VI
# Oracle Instance Architecture

This part describes the basic structural architecture of the Oracle database instance.

This part contains the following chapters:

- Oracle Database Instance
- Memory Architecture
- Process Architecture
- Application and Oracle Net Services Architecture

- Oracle Database Instance
  This chapter explains the nature of an Oracle database instance, the parameter and diagnostic files associated with an instance, and what occurs during instance creation and the opening and closing of a database.
- Memory Architecture
  This chapter discusses the memory architecture of a database instance.
- Process Architecture
  This chapter discusses the processes in an Oracle database.
- Application and Oracle Net Services Architecture
  This chapter defines application architecture and describes how an Oracle database and database applications work in a distributed processing environment. This material applies to almost every type of Oracle Database environment.

15

# Oracle Database Instance

This chapter explains the nature of an Oracle database instance, the parameter and diagnostic files associated with an instance, and what occurs during instance creation and the opening and closing of a database.

> **Note:**
>
> For the purposes of this chapter, the term "database" refers to a multitenant container database (CDB) unless otherwise noted.

This chapter contains the following sections:


- Introduction to the Oracle Database Instance
- Overview of Database Instance Startup and Shutdown
- Overview of Checkpoints
- Overview of Instance Recovery
- Overview of Parameter Files
- Overview of Diagnostic Files


- Introduction to the Oracle Database Instance
  A **database instance** is a set of memory structures that manage database files.
- Overview of Database Instance Startup and Shutdown
  A **database instance** provides user access to a database. The instance and the database can be in various states.
- Overview of Checkpoints
  A **checkpoint** is a crucial mechanism in consistent database shutdowns, instance recovery, and Oracle Database operation generally.
- Overview of Instance Recovery
  **Instance recovery** is the process of applying records in the online redo log to data files to reconstruct changes made after the most recent checkpoint.
- Overview of Parameter Files
  To start a database instance, Oracle Database must read either a **server parameter file**, which is recommended, or a **text initialization parameter file**, which is a legacy implementation. These files contain a list of configuration parameters.
- Overview of Diagnostic Files
  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.

## Introduction to the Oracle Database Instance

A **database instance** is a set of memory structures that manage database files.

At the physical level, a CDB is a set of files on disk created by the `CREATE DATABASE` statement. A CDB contains one or more user-created PDBs. A PDB contains its own set of data files within the overall set of data files that belongs to the CDB. The database instance manages the data associated with the CDB and its PDBs and serves their users.

Every running CDB is associated with at least one Oracle database instance. Because an instance exists in memory and a database (in the narrowest sense of term) is a set of files on disk, an instance can exist without a database and a database can exist without an instance.

- Database Instance Structure
  When an instance is started, Oracle Database allocates a memory area called the **system global area (SGA)** and starts one or more **background processes**.
- Database Instance Configurations
  Oracle Database runs in either a single-instance configuration or an Oracle Real Application Clusters (Oracle RAC) configuration. These configurations are mutually exclusive.
- Read/Write and Read-Only Instances
  Every database instance is either read/write or read-only.
- Duration of a Database Instance
  A database instance begins when it is created with the `STARTUP` command and ends when it is terminated.
- Identification of a Database Instance
  Multiple database instances can reside on a single host. Therefore, you must have a means of specifying which instance you want to access.

## Database Instance Structure

When an instance is started, Oracle Database allocates a memory area called the **system global area (SGA)** and starts one or more **background processes**.

The SGA serves various purposes, including the following:

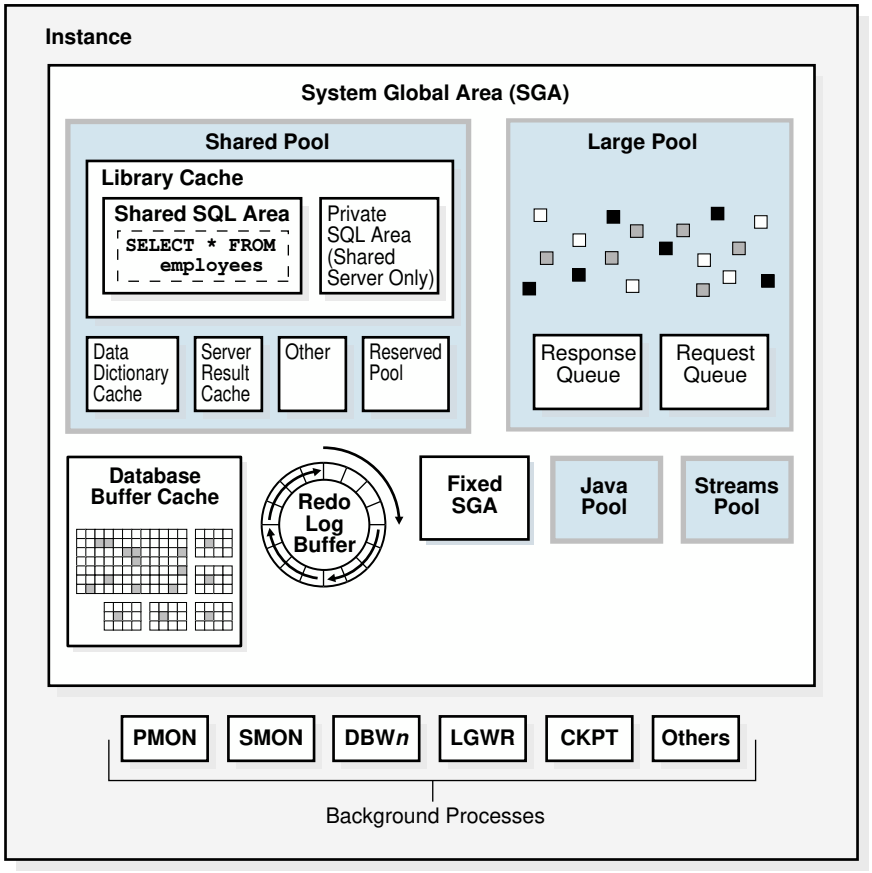
- Maintaining internal data structures that many processes and threads access concurrently
- Caching data blocks read from disk
- Buffering redo data before writing it to the online redo log files
- Storing SQL execution plans

Oracle processes running on a single computer share the SGA. The way in which Oracle processes associate with the SGA varies according to operating system.

A database instance also includes background processes and server processes. Each process has its own dedicated memory considered as part of the instance. The instance continues to function when server processes terminate.

The following graphic shows the main components of an Oracle database instance.

**Figure 15-1 Database Instance**




**See Also:**

- "Overview of the System Global Area (SGA)"
- "Overview of Background Processes"

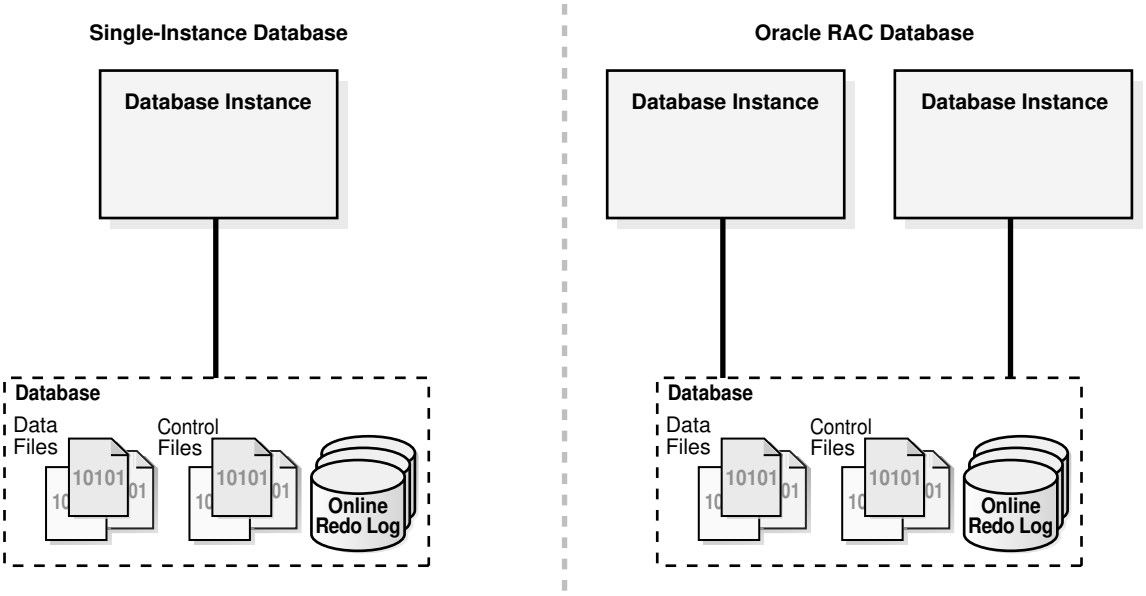
## Database Instance Configurations

Oracle Database runs in either a single-instance configuration or an Oracle Real Application Clusters (Oracle RAC) configuration. These configurations are mutually exclusive.

In a single-instance configuration, a one-to-one relationship exists between the database and a database instance. In Oracle RAC, a one-to-many relationship exists between the database and database instances.

The following figure shows possible database instance configurations.

**Figure 15-2 Database Instance Configurations**




Whether in a single-instance or Oracle RAC configuration, a database instance is associated with only one database at a time. You can start a database instance and **mount** (associate the instance with) one database, but not mount two databases simultaneously with the same instance.

> **Note:**
>
> This chapter discusses a single-instance database configuration unless otherwise noted.

Multiple instances can run concurrently on the same computer, each accessing its own database. For example, a computer can host two distinct databases: `prod1` and `prod2`. One database instance manages `prod1`, while a separate instance manages `prod2`.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for information specific to Oracle RAC

## Read/Write and Read-Only Instances

Every database instance is either read/write or read-only.

A read/write database instance, which is the default, can process DML and supports direct connections from client applications. In contrast, a read-only database instance can process queries, but does not support modification DML (`UPDATE`, `DELETE`, `INSERT`, and `MERGE`) or direct client connections.

> **Note:**
>
> Unless stated otherwise in this manual, all references to database instances are to read/write instances.

In previous releases, all database instances—unless they accessed a standby database—were read/write. Starting in Oracle Database 12c Release 2 (12.2), read-only and read/write instances can coexist within a single database. This configuration is useful for parallel SQL statements that both query and modify data, because both read/write and read-only instances can query, while the read/write instances modify.

Unlike read/write instances, read-only instances have the following characteristics:

- Can only open a database that has already been opened by a read/write instance
- Disable many background processes, including the checkpoint and archiver processes, which are not necessary
- Can mount a disabled redo thread or a thread without any online redo log

To designate an instance as read-only, set the `INSTANCE_MODE` initialization parameter to `READ_ONLY`. The default value of the parameter is `READ_WRITE`.

> **See Also:**
>
> - "Overview of Background Processes" to learn more about the checkpoint and archiver background processes
> - "Overview of the Online Redo Log"
> - *Oracle Database Reference* to learn more about the `INSTANCE_MODE` initialization parameter

## Duration of a Database Instance

A database instance begins when it is created with the `STARTUP` command and ends when it is terminated.

During this period, a database instance can associate itself with one and only one database. Furthermore, the instance can mount a database only once, close it only once, and open it only once. After a database has been closed or shut down, you must start a *different* instance to mount and open this database.

The following table illustrates a database instance attempting to reopen a database that it previously closed.

**Table 15-1 Duration of an Instance**

| Statement | Explanation |
|---|---|
| SQL> STARTUP<br>ORACLE instance started.<br><br>Total System Global Area 468729856 bytes<br>Fixed Size 1333556 bytes<br>Variable Size 440403660 bytes<br>Database Buffers 16777216 bytes<br>Redo Buffers 10215424 bytes<br>Database mounted.<br>Database opened. | The STARTUP command creates an instance, which mounts and opens the database. |
| SQL> SELECT<br>TO_CHAR(STARTUP_TIME,'MON-DD-RR HH24:MI:SS')<br>AS "Inst Start Time" FROM V$INSTANCE;<br><br>Inst Start Time<br>------------------<br>JUN-18-14 **13:14:48** | This query shows the time that the current instance was started. |
| SQL> SHUTDOWN IMMEDIATE | The instance closes the database and shuts down, ending the life of this instance. |
| SQL> STARTUP<br>Oracle instance started.<br>. . . | The STARTUP command creates a new instance and mounts and open the database. |
| SQL> SELECT<br>TO_CHAR(STARTUP_TIME, 'MON-DD-RR HH24:MI:SS')<br>AS "Inst Start Time" FROM V$INSTANCE;<br><br>Inst Start Time<br>------------------<br>JUN-18-14 **13:16:40** | This query shows the time that the current instance was started. The different start time shows that this instance is different from the one that shut down the database. |

# Identification of a Database Instance

Multiple database instances can reside on a single host. Therefore, you must have a means of specifying which instance you want to access.

Oracle Optimal Flexible Architecture (OFA) rules are a set of configuration guidelines created to ensure well-organized Oracle installations. The examples in this section assume an OFA architecture.

This section contains the following topics:


- Oracle Base Directory
- Oracle Home Directory
- Oracle System Identifier (SID)


- Oracle Base Directory
  The Oracle base directory stores the binaries for Oracle products.
- Oracle Home Directory
  The **Oracle home** is the software location for an Oracle database.
- Oracle System Identifier (SID)
  The **system identifier (SID)** is a unique name for an Oracle database instance on a specific host.

> **See Also:**
>
> *Oracle Database Installation Guide for Linux* for an overview of the OFA

## Oracle Base Directory

The Oracle base directory stores the binaries for Oracle products.

The Oracle Base directory is the database home directory for Oracle Database installation owners. There can be many Oracle Database installations on a host, and many Oracle Database software installation owners.

The following example shows the Oracle base directory of the operating system user account `oracle`:

```
/u01/app/oracle
```

In the preceding path, `/u01/` is the mount point, and `/u01/app/` is the subtree for application software.

> **See Also:**
>
> *Oracle Database Installation Guide for Linux* for an overview of the Oracle base

## Oracle Home Directory

The **Oracle home** is the software location for an Oracle database.

You must specify a new Oracle home directory for each new installation of Oracle Database software. Starting in Oracle Database 21c, the Oracle home is read-only. The Oracle home stores static files such as binaries.

By default, the Oracle home directory is a descendent within the Oracle base (`ORACLE_BASE`) directory tree. You can install this release, or earlier releases of the database software, more

than once on the same host, in different Oracle home directories within a single Oracle base. Multiple databases, of different versions and owned by different user accounts, can coexist concurrently.

The following example shows the full path names of three different Oracle homes, all within the same Oracle base directory of `/u01/app/oracle/`:

```
/u01/app/oracle/product/19.3.0/dbhome_1
/u01/app/oracle/product/19.3.0/dbhome_2
/u01/app/oracle/product/20.1.0/dbhome_1
```

The part of the path name after the Oracle base (`/u01/app/oracle/`) includes the product release number (for example, `19.3.0`) and Oracle home relative directory (for example, `dbhome_1`). The `/u01/app/oracle/product/19.3.0/` directory contains two separate Oracle homes: `dbhome_1` and `dbhome_2`.

The Oracle base home (`ORACLE_BASE_HOME`) directory, which is located in `ORACLE_BASE/homes/home_name`, stores dynamic files specific to an Oracle home. The Oracle base configuration directory (`ORACLE_BASE_CONFIG`), which is shared by all Oracle homes in an Oracle base, stores instance-specific dynamic files.

In the following example, the first path is an Oracle home, and the second path is the Oracle base home for this Oracle home:

```
/u01/app/oracle/product/20.1.0/dbhome_1
/u01/app/oracle/homes/dbhome_1
```

> **See Also:**
>
> *Oracle Database Installation Guide for Linux* for an overview of the Oracle home

## Oracle System Identifier (SID)

The **system identifier (SID)** is a unique name for an Oracle database instance on a specific host.

On UNIX and Linux, Oracle Database uses the SID and Oracle home values to create a key to shared memory. Also, Oracle Database uses the SID by default to locate the initialization parameter file, which locates relevant files such as the database control files.

On most platforms, the `ORACLE_SID` environment variable sets the SID, and the `ORACLE_HOME` variable sets the Oracle home. When connecting to a database instance, clients can specify the SID in an Oracle Net connection or use a net service name. Oracle Database converts a net service name to an `ORACLE_HOME` and `ORACLE_SID`.

Instance-specific files are stored separately in the Oracle base. Files whose names include the SID reside in the `dbs` subdirectory of the Oracle base configuration directory (`ORACLE_BASE_CONFIG`). Because of this separation, you can create a database using software in an existing Oracle home, and then start an instance for this database using software that resides in a new Oracle home.

> **See Also:**
>
> - "Service Names"
> - *Oracle Database Administrator's Guide* to learn how to specify an Oracle SID

# Overview of Database Instance Startup and Shutdown

A **database instance** provides user access to a database. The instance and the database can be in various states.

- Overview of Instance and Database Startup
  Typically, you manually start an instance, and then mount and open the database, making it available for users. You can use the SQL*Plus `STARTUP` command, Oracle Enterprise Manager (Enterprise Manager), or the SRVCTL utility to perform these steps.
- Overview of Database and Instance Shutdown
  In a typical use case, you manually shut down the database, making it unavailable for users while you perform maintenance or other administrative tasks. You can use the SQL*Plus `SHUTDOWN` command or Enterprise Manager to perform these steps.

## Overview of Instance and Database Startup

Typically, you manually start an instance, and then mount and open the database, making it available for users. You can use the SQL*Plus `STARTUP` command, Oracle Enterprise Manager (Enterprise Manager), or the SRVCTL utility to perform these steps.

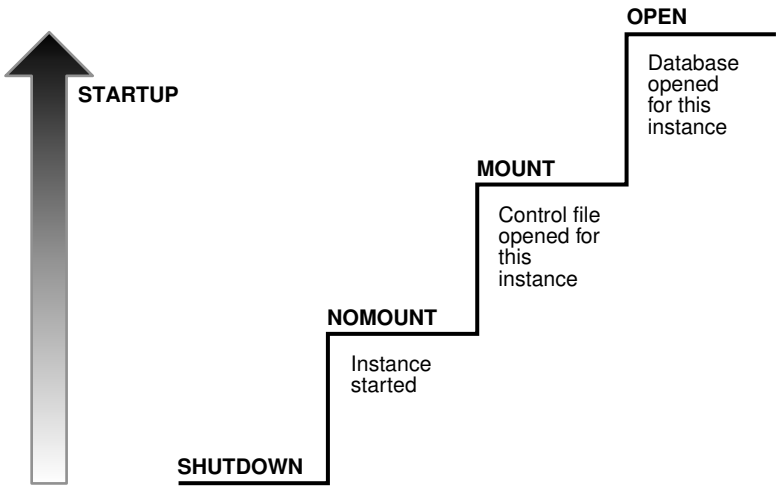
To start a database instance using Oracle Net, the following must be true:

- The database is statically registered with an Oracle Net listener.
- Your client is connected to the database with the `SYSDBA` privilege.

The listener creates a dedicated server, which can start the database instance.

The following graphic shows the database progressing from a shutdown state to an open state.

**Figure 15-3 Instance and Database Startup Sequence**



A database goes through the following phases when it proceeds from a shutdown state to an open database state.

**Table 15-2 Steps in Instance Startup**

| Phase | Mount State | Description | To Learn More |
|---|---|---|---|
| 1 | Instance started without mounting database | The instance is started, but is not yet associated with a database. | "How an Instance Is Started" |
| 2 | Database mounted | The instance is started and is associated with a database by reading its control file. The database is closed to users. | "How a Database Is Mounted" |
| 3 | Database open | The instance is started and is associated with an open database. The data contained in the data files is accessible to authorized users. | "How a Database Is Opened" |

- Connection with Administrator Privileges
  Database startup and shutdown are powerful administrative options that are restricted to users who connect to Oracle Database with administrator privileges.
- How an Instance Is Started
  When Oracle Database starts an instance, it proceeds through stages.
- How a Database Is Mounted
  The instance **mounts** a database to associate the database with this instance.
- How a Database Is Opened
  Opening a mounted database makes it available for normal database operation.

**See Also:**

- "The Oracle Net Listener"
- "Overview of Control Files"
- *Oracle Database Administrator's Guide* to learn how to start an instance
- *Oracle Database Administrator's Guide* to learn how to use SRVCTL

## Connection with Administrator Privileges

Database startup and shutdown are powerful administrative options that are restricted to users who connect to Oracle Database with administrator privileges.

Normal users do not have control over the current status of an Oracle database. Depending on the operating system, one of the following conditions establishes administrator privileges for a user:

- The operating system privileges of the user enable him or her to connect using administrator privileges.
- The user is granted special system privileges, and the database uses password files to authenticate database administrators over the network.

The following special system privileges enable access to a database instance even when the database is not open:

- SYSDBA
- SYSOPER
- SYSBACKUP
- SYSDG
- SYSKM

Control of the preceding privileges is outside of the database itself. When you connect to a database with the SYSDBA system privilege, you are in the schema owned by SYS. When you connect as SYSOPER, you are in the public schema. SYSOPER privileges are a subset of SYSDBA privileges.

> **See Also:**
>
> - "SYS and SYSTEM Accounts"
> - *Oracle Database Security Guide* to learn about predefined administrative accounts
> - *Oracle Database Administrator's Guide* to learn about system privileges
> - *Oracle Database Installation Guide* to learn more about operating system privilege groups

## How an Instance Is Started

When Oracle Database starts an instance, it proceeds through stages.

The stages are as follows:

1. Searches for a server parameter file in a platform-specific default location and, if not found, for a text initialization parameter file (specifying STARTUP with the SPFILE or PFILE parameters overrides the default behavior)
2. Reads the parameter file to determine the values of initialization parameters
3. Allocates the SGA based on the initialization parameter settings
4. Starts the Oracle background processes
5. Opens the alert log and trace files and writes all explicit parameter settings to the alert log in valid parameter syntax

At this stage, no database is associated with the instance. Scenarios that require a NOMOUNT state include database creation and certain backup and recovery operations.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage initialization parameters using a server parameter file

## How a Database Is Mounted

The instance **mounts** a database to associate the database with this instance.

To mount the database, the instance obtains the names of the database control files specified in the `CONTROL_FILES` initialization parameter and opens the files. Oracle Database reads the control files to find the names of the data files and the online redo log files that it will attempt to access when opening the database.

In a mounted database, the database is closed and accessible only to database administrators. Administrators can keep the database closed while completing specific maintenance operations. However, the database is not available for normal operations.

If Oracle Database allows multiple instances to mount the same database concurrently, then the `CLUSTER_DATABASE` initialization parameter setting can make the database available to multiple instances. Database behavior depends on the setting:

- If `CLUSTER_DATABASE` is `false` (default) for the first instance that mounts a database, then only this instance can mount the database.
- If `CLUSTER_DATABASE` is `true` for the first instance, then other instances can mount the database if their `CLUSTER_DATABASE` parameter settings are set to `true`. The number of instances that can mount the database is subject to a predetermined maximum specified when creating the database.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to mount a database
> - *Oracle Real Application Clusters Administration and Deployment Guide* for more information about the use of multiple instances with a single database

## How a Database Is Opened

Opening a mounted database makes it available for normal database operation.

Any valid user can connect to an open database and access its information. Usually, a database administrator opens the database to make it available for general use.

When you open the database, Oracle Database performs the following actions:

- Opens the online data files in tablespaces other than undo tablespaces

  If a tablespace was offline when the database was previously shut down, then the tablespace and its corresponding data files will be offline when the database reopens.
- Acquires an undo tablespace

  If multiple undo tablespaces exists, then the `UNDO_TABLESPACE` initialization parameter designates the undo tablespace to use. If this parameter is not set, then the first available undo tablespace is chosen.
- Opens the online redo log files

- Read-Only Mode
  By default, the database opens in **read/write mode**. In this mode, users can make changes to the data, generating redo in the online redo log. Alternatively, you can open in **read-only mode** to prevent data modification by user transactions.
- Database File Checks
  If any of the data files or redo log files are not present when the instance attempts to open the database, or if the files are present but fail consistency tests, then the database returns an error. Media recovery may be required.

> **See Also:**
>
> - "Online and Offline Tablespaces"
> - *Oracle Database Backup and Recovery User's Guide* to learn more about data repair

## Read-Only Mode

By default, the database opens in **read/write mode**. In this mode, users can make changes to the data, generating redo in the online redo log. Alternatively, you can open in **read-only mode** to prevent data modification by user transactions.

> **Note:**
>
> By default, a physical standby database opens in read-only mode.

Read-only mode restricts database access to read-only transactions, which cannot write to data files or to online redo log files. However, the database can perform recovery or operations that change the database state without generating redo. For example, in read-only mode:

- Data files can be taken offline and online. However, you cannot take permanent tablespaces offline.
- Offline data files and tablespaces can be recovered.
- The control file remains available for updates about the state of the database.
- Temporary tablespaces created with the `CREATE TEMPORARY TABLESPACE` statement are read/write.
- Writes to operating system audit trails, trace files, and alert logs can continue.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to open a database in read-only mode
> - *Oracle Data Guard Concepts and Administration*

### Database File Checks

If any of the data files or redo log files are not present when the instance attempts to open the database, or if the files are present but fail consistency tests, then the database returns an error. Media recovery may be required.

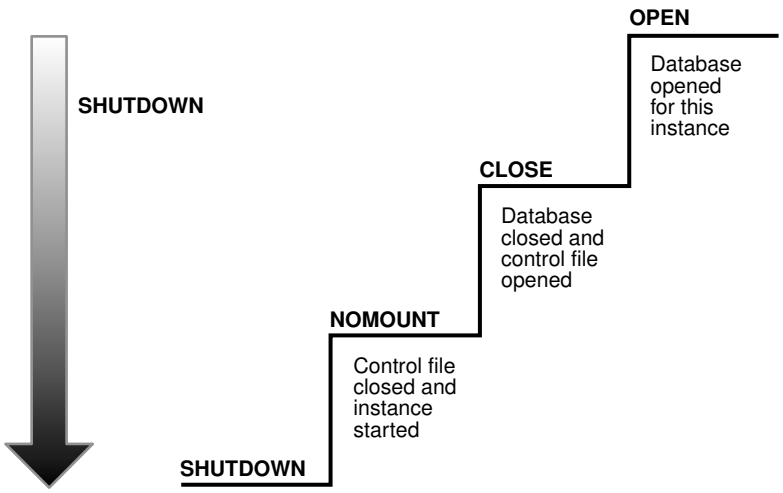
> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for an overview of backup and recovery

## Overview of Database and Instance Shutdown

In a typical use case, you manually shut down the database, making it unavailable for users while you perform maintenance or other administrative tasks. You can use the SQL*Plus `SHUTDOWN` command or Enterprise Manager to perform these steps.

The following figure shows the progression from an open state to a consistent shutdown.

**Figure 15-4 Instance and Database Shutdown Sequence**




Oracle Database automatically performs the following steps whenever an open database is shut down consistently.

**Table 15-3 Steps in Consistent Shutdown**

| Phase | Mount State | Description | To Learn More |
|---|---|---|---|
| 1 | Database closed | The database is mounted, but online data files and redo log files are closed. | "How a Database Is Closed" |

**Table 15-3 (Cont.) Steps in Consistent Shutdown**

| Phase | Mount State | Description | To Learn More |
|---|---|---|---|
| 2 | Database unmounted | The instance is started, but is no longer associated with the control file of the database. | "How a Database Is Unmounted" |
| 3 | Database instance shut down | The database instance is no longer started. | "How an Instance Is Shut Down" |

Oracle Database does not go through all of the preceding steps in an instance failure or `SHUTDOWN ABORT`, which immediately terminates the instance.

- Shutdown Modes
  A database administrator with `SYSDBA` or `SYSOPER` privileges can shut down the database using the SQL*Plus `SHUTDOWN` command or Enterprise Manager. The `SHUTDOWN` command has options that determine shutdown behavior.
- How a Database Is Closed
  The database close operation is implicit in a database shutdown. The nature of the operation depends on whether the database shutdown is normal or abnormal.
- How a Database Is Unmounted
  After the database is closed, Oracle Database unmounts the database to disassociate it from the instance.
- How an Instance Is Shut Down
  The final step in database shutdown is shutting down the instance. When the database instance shuts down, the SGA ceases to occupy memory, and the background processes terminate.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to shut down a database

## Shutdown Modes

A database administrator with `SYSDBA` or `SYSOPER` privileges can shut down the database using the SQL*Plus `SHUTDOWN` command or Enterprise Manager. The `SHUTDOWN` command has options that determine shutdown behavior.

The following table summarizes the behavior of the different shutdown modes.

**Table 15-4 Database Shutdown Modes**

| Database Behavior | ABORT | IMMEDIATE | TRANSACTIONAL | NORMAL |
|---|---|---|---|---|
| Permits new user connections | No | No | No | No |
| Waits until current sessions end | No | No | No | Yes |
| Waits until current transactions end | No | No | Yes | Yes |

**Table 15-4 (Cont.) Database Shutdown Modes**

| Database Behavior | ABORT | IMMEDIATE | TRANSACTIONAL | NORMAL |
|---|---|---|---|---|
| Performs a checkpoint and closes open files | No | Yes | Yes | Yes |

The possible `SHUTDOWN` statements are:

- `SHUTDOWN ABORT`

  This mode is intended for emergency situations, such as when no other form of shutdown is successful. This mode of shutdown is the fastest. However, a subsequent open of this database may take substantially longer because instance recovery must be performed to make the data files consistent.

  Because `SHUTDOWN ABORT` does not checkpoint the open data files, instance recovery is necessary before the database can reopen. The other shutdown modes do not require instance recovery before the database can reopen.

  > **Note:**
  >
  > In a PDB, issuing `SHUTDOWN ABORT` is equivalent to issuing `SHUTDOWN IMMEDIATE` at the CDB level.

- `SHUTDOWN IMMEDIATE`

  This mode is typically the fastest next to `SHUTDOWN ABORT`. Oracle Database terminates any executing SQL statements and disconnects users. Active transactions are terminated and uncommitted changes are rolled back.

- `SHUTDOWN TRANSACTIONAL`

  This mode prevents users from starting new transactions, but waits for all current transactions to complete before shutting down. This mode can take a significant amount of time depending on the nature of the current transactions.

- `SHUTDOWN NORMAL`

  This is the default mode of shutdown. The database waits for all connected users to disconnect before shutting down.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about the different shutdown modes
> - *SQL*Plus User's Guide and Reference* to learn about the `SHUTDOWN` command

## How a Database Is Closed

The database close operation is implicit in a database shutdown. The nature of the operation depends on whether the database shutdown is normal or abnormal.

- How a Database Is Closed During Normal Shutdown
  When a database is closed as part of a SHUTDOWN with any option other than ABORT, Oracle Database writes data in the SGA to the data files and online redo log files.
- How a Database Is Closed During Abnormal Shutdown
  If a SHUTDOWN ABORT or abnormal termination occurs, then the instance of an open database closes and shuts down the database instantaneously.

## How a Database Is Closed During Normal Shutdown

When a database is closed as part of a SHUTDOWN with any option other than ABORT, Oracle Database writes data in the SGA to the data files and online redo log files.

Afterward, the database closes online data files and online redo log files. Any offline data files of offline tablespaces have been closed already. When the database reopens, any tablespace that was offline remains offline.

At this stage, the database is closed and inaccessible for normal operations. The control files remain open after a database is closed.

## How a Database Is Closed During Abnormal Shutdown

If a SHUTDOWN ABORT or abnormal termination occurs, then the instance of an open database closes and shuts down the database instantaneously.

In an abnormal shutdown, Oracle Database does not write data in the buffers of the SGA to the data files and redo log files. The subsequent reopening of the database requires instance recovery, which Oracle Database performs automatically.

# How a Database Is Unmounted

After the database is closed, Oracle Database unmounts the database to disassociate it from the instance.

After a database is unmounted, Oracle Database closes the control files of the database. At this point, the database instance remains in memory.

# How an Instance Is Shut Down

The final step in database shutdown is shutting down the instance. When the database instance shuts down, the SGA ceases to occupy memory, and the background processes terminate.

In unusual circumstances, shutdown of a database instance may not occur cleanly. Memory structures may not be removed from memory or one of the background processes may not be terminated. When remnants of a previous instance exist, a subsequent instance startup may fail. In such situations, you can force the new instance to start by removing the remnants of the previous instance and then starting a new instance, or by issuing a SHUTDOWN ABORT statement.

In some cases, process cleanup itself can encounter errors, which can result in the termination of process monitor (PMON) or the instance. The dynamic initialization parameter INSTANCE_ABORT_DELAY_TIME specifies how many seconds to delay an internally generated instance failure. This delay gives you a chance to respond. The database writes a message to the alert log when the delayed termination is initiated. In some circumstances, by allowing certain database resources to be quarantined, the instance can avoid termination.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more detailed information about database shutdown
> - *Oracle Database Reference* to learn more about the `INSTANCE_ABORT_DELAY_TIME` initialization parameter

# Overview of Checkpoints

A **checkpoint** is a crucial mechanism in consistent database shutdowns, instance recovery, and Oracle Database operation generally.

The term has the following related meanings:

- A data structure that indicates the **checkpoint position**, which is the SCN in the redo stream where instance recovery must begin

  The checkpoint position is determined by the oldest dirty buffer in the database buffer cache. The checkpoint position acts as a pointer to the redo stream and is stored in the control file and in each data file header.
- The writing of modified database buffers in the database buffer cache to disk

- Purpose of Checkpoints
  Oracle Database uses checkpoints to achieve multiple goals.
- When Oracle Database Initiates Checkpoints
  The **checkpoint process (CKPT)** is responsible for writing checkpoints to the data file headers and control file.

> **See Also:**
>
> "System Change Numbers (SCNs)"

## Purpose of Checkpoints

Oracle Database uses checkpoints to achieve multiple goals.

Goals include the following:

- Reduce the time required for recovery in case of an instance or media failure
- Ensure that the database regularly writes dirty buffers in the buffer cache to disk
- Ensure that the database writes all committed data to disk during a consistent shutdown

## When Oracle Database Initiates Checkpoints

The **checkpoint process (CKPT)** is responsible for writing checkpoints to the data file headers and control file.

Checkpoints occur in a variety of situations. For example, Oracle Database uses the following types of checkpoints:

- Thread checkpoints

  The database writes to disk all buffers modified by redo in a specific thread before a certain target. The set of thread checkpoints on all instances in a database is a **database checkpoint**. Thread checkpoints occur in the following situations:

  - Consistent database shutdown
  - `ALTER SYSTEM CHECKPOINT` statement
  - Online redo log switch
  - `ALTER DATABASE BEGIN BACKUP` statement

- Tablespace and data file checkpoints

  The database writes to disk all buffers modified by redo before a specific target. A tablespace checkpoint is a set of data file checkpoints, one for each data file in the tablespace. These checkpoints occur in a variety of situations, including making a tablespace read-only or taking it offline normal, shrinking a data file, or executing `ALTER TABLESPACE BEGIN BACKUP`.

- Incremental checkpoints

  An incremental checkpoint is a type of thread checkpoint partly intended to avoid writing large numbers of blocks at online redo log switches. DBW checks at least every three seconds to determine whether it has work to do. When DBW writes dirty buffers, it advances the checkpoint position, causing CKPT to write the checkpoint position to the control file, but not to the data file headers.

Other types of checkpoints include instance and media recovery checkpoints and checkpoints when schema objects are dropped or truncated.

> **See Also:**
>
> - "Checkpoint Process (CKPT)"
> - *Oracle Real Application Clusters Administration and Deployment Guide* for information about global checkpoints in Oracle RAC

# Overview of Instance Recovery

**Instance recovery** is the process of applying records in the online redo log to data files to reconstruct changes made after the most recent checkpoint.

Instance recovery occurs automatically when an administrator attempts to open a database that was previously shut down inconsistently.

- Purpose of Instance Recovery
  Instance recovery ensures that the database is in a consistent state after an instance failure. The files of a database can be left in an inconsistent state because of how Oracle Database manages database changes.
- When Oracle Database Performs Instance Recovery
  Whether instance recovery is required depends on the state of the redo threads.

- Importance of Checkpoints for Instance Recovery
  Instance recovery uses checkpoints to determine which changes must be applied to the data files. The checkpoint position guarantees that every committed change with an SCN *lower than* the checkpoint SCN is saved to the data files.
- Instance Recovery Phases
  The first phase of instance recovery is called **cache recovery** or **rolling forward**, and reapplies all changes recorded in the online redo log to the data files.

## Purpose of Instance Recovery

Instance recovery ensures that the database is in a consistent state after an instance failure. The files of a database can be left in an inconsistent state because of how Oracle Database manages database changes.

A redo thread is a record of all of the changes generated by an instance. A single-instance database has one thread of redo, whereas an Oracle RAC database has multiple redo threads, one for each database instance.

When a transaction is committed, log writer process (LGWR) writes both the remaining redo entries in memory and the transaction SCN to the online redo log. However, the database writer (DBW) process writes modified data blocks to the data files whenever it is most efficient. For this reason, uncommitted changes may temporarily exist in the data files while committed changes do not yet exist in the data files.

If an instance of an open database fails, either because of a `SHUTDOWN ABORT` statement or abnormal termination, then the following situations can result:

- Data blocks committed by a transaction are not written to the data files and appear only in the online redo log. These changes must be reapplied to the data files.
- The data files contains changes that had not been committed when the instance failed. These changes must be rolled back to ensure transactional consistency.

Instance recovery uses only online redo log files and current online data files to synchronize the data files and ensure that they are consistent.

> **See Also:**
>
> - "Database Writer Process (DBW)" and "Database Buffer Cache"
> - "Introduction to Data Concurrency and Consistency"

## When Oracle Database Performs Instance Recovery

Whether instance recovery is required depends on the state of the redo threads.

A redo thread is marked open in the control file when a database instance opens in read/write mode, and is marked closed when the instance is shut down consistently. If redo threads are marked open in the control file, but no live instances hold the thread enqueues corresponding to these threads, then the database requires instance recovery.

Oracle Database performs instance recovery automatically in the following situations:

- The database opens for the first time after the failure of a single-instance database or all instances of an Oracle RAC database. This form of instance recovery is also called **crash**

**recovery**. Oracle Database recovers the online redo threads of the terminated instances together.

- Some but not all instances of an Oracle RAC database fail. Instance recovery is performed automatically by a surviving instance in the configuration.

The SMON background process performs instance recovery, applying online redo automatically. No user intervention is required.

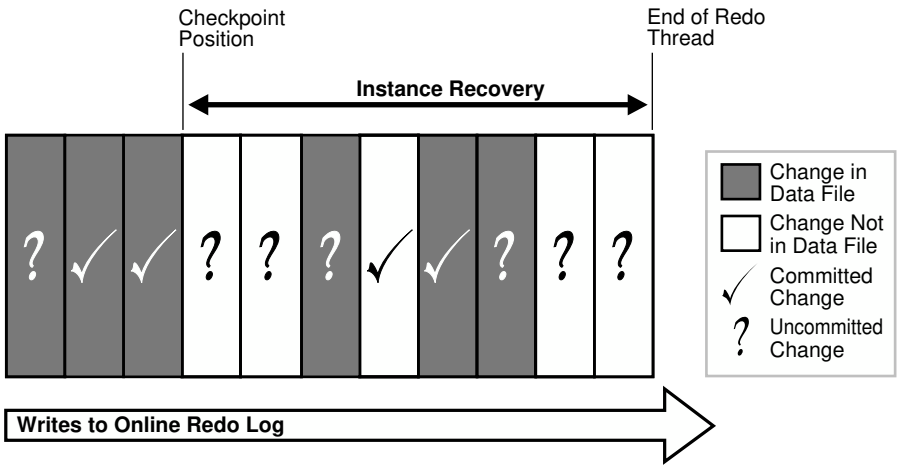
> **See Also:**
>
> - "System Monitor Process (SMON)"
> - *Oracle Real Application Clusters Administration and Deployment Guide* to learn about instance recovery in an Oracle RAC database

## Importance of Checkpoints for Instance Recovery

Instance recovery uses checkpoints to determine which changes must be applied to the data files. The checkpoint position guarantees that every committed change with an SCN *lower than* the checkpoint SCN is saved to the data files.

The following figure depicts the redo thread in the online redo log.

**Figure 15-5 Checkpoint Position in Online Redo Log**




During instance recovery, the database must apply the changes that occur between the checkpoint position and the end of the redo thread. As shown in Figure 15-5, some changes may already have been written to the data files. However, only changes with SCNs lower than the checkpoint position are *guaranteed* to be on disk.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* to learn how to limit instance recovery time

## Instance Recovery Phases

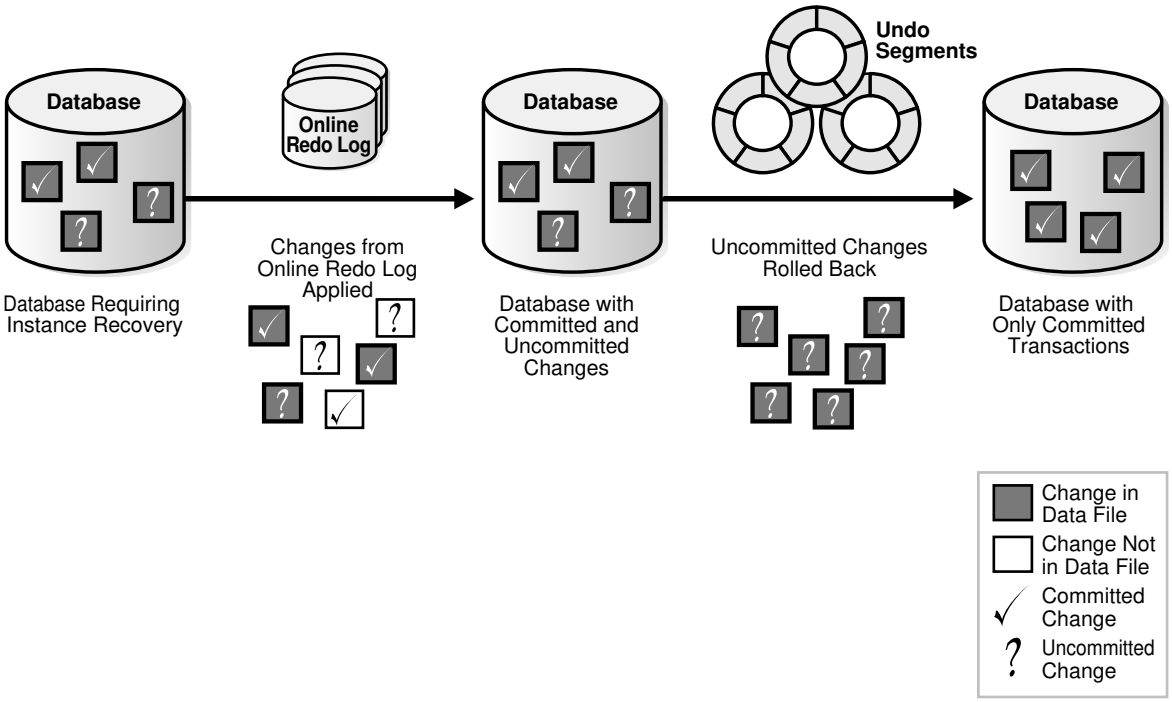
The first phase of instance recovery is called **cache recovery** or **rolling forward**, and reapplies all changes recorded in the online redo log to the data files.

Because the online redo log contains undo data, rolling forward also regenerates the corresponding undo segments. Rolling forward proceeds through as many online redo log files as necessary to bring the database forward in time. After rolling forward, the data blocks contain all committed changes recorded in the online redo log files. These files could also contain uncommitted changes that were either saved to the data files before the failure, or were recorded in the online redo log and introduced during cache recovery.

After the roll forward, any changes that were not committed must be undone. Oracle Database uses the checkpoint position, which guarantees that every committed change with an SCN lower than the checkpoint SCN is saved on disk. Oracle Database applies undo blocks to roll back uncommitted changes in data blocks that were written before the failure or introduced during cache recovery. This phase is called **rolling back** or transaction recovery.

The following figure illustrates rolling forward and rolling back, the two steps necessary to recover from database instance failure.

**Figure 15-6 Basic Instance Recovery Steps: Rolling Forward and Rolling Back**




Oracle Database can roll back multiple transactions simultaneously as needed. All transactions that were active at the time of failure are marked as terminated. Instead of waiting for the

SMON process to roll back terminated transactions, new transactions can roll back individual blocks themselves to obtain the required data.

> **See Also:**
>
> - "Undo Segments" to learn more about undo data
> - *Oracle Database Performance Tuning Guide* for a discussion of instance recovery mechanics and tuning

# Overview of Parameter Files

To start a database instance, Oracle Database must read either a **server parameter file**, which is recommended, or a **text initialization parameter file**, which is a legacy implementation. These files contain a list of configuration parameters.

To create a database manually, you must start an instance with a parameter file and then issue a `CREATE DATABASE` statement. Thus, the instance and parameter file can exist even when the database itself does not exist.

- Initialization Parameters
  **Initialization parameters** are configuration parameters that affect the basic operation of an instance. The instance reads initialization parameters from a file at startup.
- Server Parameter Files
  A **server parameter file** is a repository for initialization parameters.
- Text Initialization Parameter Files
  A **text initialization parameter file** is a text file that contains a list of initialization parameters.
- Modification of Initialization Parameter Values
  You can adjust initialization parameters to modify the behavior of a database. The classification of parameters as **static** or **dynamic** determines how they can be modified.

## Initialization Parameters

**Initialization parameters** are configuration parameters that affect the basic operation of an instance. The instance reads initialization parameters from a file at startup.

Oracle Database provides many initialization parameters to optimize its operation in diverse environments. Only a few of these parameters must be explicitly set because the default values are usually adequate.

- Functional Groups of Initialization Parameters
  Initialization parameters fall into different functional groups.
- Basic and Advanced Initialization Parameters
  Initialization parameters are divided into two groups: basic and advanced.

### Functional Groups of Initialization Parameters

Initialization parameters fall into different functional groups.

Most initialization parameters belong to one of the following groups:

- Parameters that name entities such as files or directories
- Parameters that set limits for a process, database resource, or the database itself
- Parameters that affect capacity, such as the size of the SGA (these parameters are called **variable parameters**)

Variable parameters are of particular interest to database administrators because they can use these parameters to improve database performance.

## Basic and Advanced Initialization Parameters

Initialization parameters are divided into two groups: basic and advanced.

Typically, you must set and tune only the approximately 30 basic parameters to obtain reasonable performance. The basic parameters set characteristics such as the database name, locations of the control files, database block size, and undo tablespace.

In rare situations, modification to the advanced parameters may be required for optimal performance. The advanced parameters enable expert DBAs to adapt the behavior of the Oracle Database to meet unique requirements.

Oracle Database provides values in the starter initialization parameter file provided with your database software, or as created for you by the Database Configuration Assistant (DBCA). You can edit these Oracle-supplied initialization parameters and add others, depending on your configuration and how you plan to tune the database. For relevant initialization parameters not included in the parameter file, Oracle Database supplies defaults.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to specify initialization parameters
- *Oracle Database Reference* for an explanation of the types of initialization parameters
- *Oracle Database Reference* for a description of `V$PARAMETER` and *SQL*Plus User's Guide and Reference* for `SHOW PARAMETER` syntax

# Server Parameter Files

A **server parameter file** is a repository for initialization parameters.

A server parameter file has the following key characteristics:

- Only Oracle Database reads and writes to the server parameter file.
- Only one server parameter file exists for a database. This file must reside on the database host.
- The server parameter file is binary and cannot be modified by a text editor.
- Initialization parameters stored in the server parameter file are persistent. Any changes made to the parameters while a database instance is running can persist across instance shutdown and startup.

A server parameter file eliminates the need to maintain multiple text initialization parameter files for client applications. A server parameter file is initially built from a text initialization

parameter file using the `CREATE SPFILE` statement. It can also be created directly by the Database Configuration Assistant.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn more about server parameter files
> - *Oracle Database SQL Language Reference* to learn about `CREATE SPFILE`

## Text Initialization Parameter Files

A **text initialization parameter file** is a text file that contains a list of initialization parameters.

This type of parameter file, which is a legacy implementation of the parameter file, has the following key characteristics:

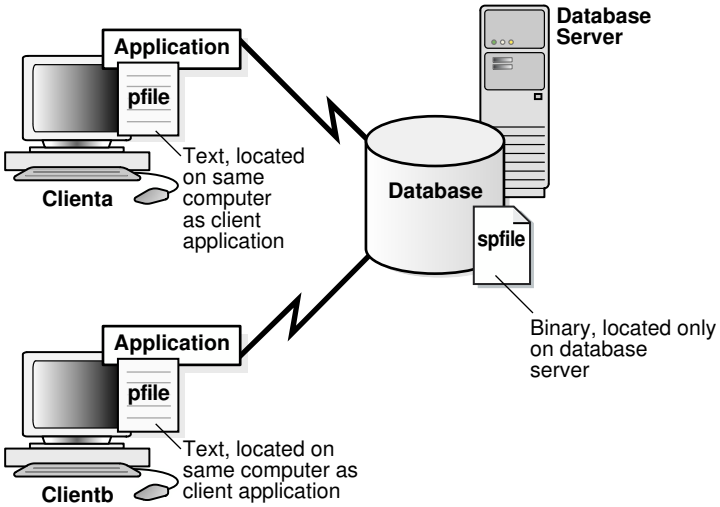
- When starting up or shutting down a database, the text initialization parameter file must reside on the same host as the client application that connects to the database.
- A text initialization parameter file is text-based, not binary.
- Oracle Database can read but not write to the text initialization parameter file. To change the parameter values you must manually alter the file with a text editor.
- Changes to initialization parameter values by `ALTER SYSTEM` are only in effect for the current instance. You must manually update the text initialization parameter file and restart the instance for the changes to be known.

The text initialization parameter file contains a series of `key=value` pairs, one per line. For example, a portion of an initialization parameter file could look as follows:

```
db_name=sample
control_files=/disk1/oradata/sample_cf.dbf
db_block_size=8192
open_cursors=52
undo_management=auto
shared_pool_size=280M
pga_aggregate_target=29M
.
.
.
```

To illustrate the manageability problems that text parameter files can create, assume that you use computers `clienta` and `clientb` and must be able to start the database with SQL*Plus on either computer. In this case, two separate text initialization parameter files must exist, one on each computer, as shown in Figure 15-7. A server parameter file solves the problem of the proliferation of parameter files.

**Figure 15-7 Multiple Initialization Parameter Files**




> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn more about text initialization parameter files
> - *Oracle Database SQL Language Reference* to learn about `CREATE PFILE`

## Modification of Initialization Parameter Values

You can adjust initialization parameters to modify the behavior of a database. The classification of parameters as **static** or **dynamic** determines how they can be modified.

The following table summarizes the differences.

**Table 15-5 Static and Dynamic Initialization Parameters**

| Characteristic | Static | Dynamic |
|---|---|---|
| Requires modification of the parameter file (text or server) | Yes | No |
| Requires database instance restart before setting takes affect | Yes | No |
| Described as "Modifiable" in *Oracle Database Reference* initialization parameter entry | No | Yes |
| Modifiable only for the database or instance | Yes | No |

Static parameters include `DB_BLOCK_SIZE`, `DB_NAME`, and `COMPATIBLE`. Dynamic parameters are grouped into **session-level parameters**, which affect only the current user session, and **system-level parameters**, which affect the database and all sessions. For example, `MEMORY_TARGET` is a system-level parameter, while `NLS_DATE_FORMAT` is a session-level parameter.

The **scope** of a parameter change depends on when the change takes effect. When an instance has been started with a server parameter file, you can use the `ALTER SYSTEM SET` statement to change values for system-level parameters as follows:

- `SCOPE=MEMORY`

  Changes apply to the database instance only. The change will not persist if the database is shut down and restarted.

- `SCOPE=SPFILE`

  Changes apply to the server parameter file but do not affect the current instance. Thus, the changes do not take effect until the instance is restarted.

  > **Note:**
  >
  > You must specify `SPFILE` when changing the value of a parameter described as not modifiable in *Oracle Database Reference*.

- `SCOPE=BOTH`

  Oracle Database writes changes both to memory and to the server parameter file. This is the default scope when the database is using a server parameter file.

The database prints the new value and the old value of an initialization parameter to the alert log. As a preventative measure, the database validates changes of basic parameter to prevent invalid values from being written to the server parameter file.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* to learn how to choose a locale with the `NLS_LANG` environment variable
> - *Oracle Database Administrator's Guide* to learn how to change initialization parameter settings
> - *Oracle Database Reference* for descriptions of all initialization parameters
> - *Oracle Database SQL Language Reference* for `ALTER SYSTEM` syntax and semantics

# Overview of Diagnostic Files

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 advanced fault diagnosability infrastructure are the following:

- Detecting problems proactively
- Limiting damage and interruptions after a problem is detected
- Reducing problem diagnostic and resolution time

- Improving manageability by enabling trace files to be partitioned, allowing user to define the size per piece and maximum number of pieces to retain, and disabling tracing after a user-specified disk space limit is reached
- Simplifying customer interaction with Oracle Support

- Automatic Diagnostic Repository
  **Automatic Diagnostic Repository (ADR)** is a file-based repository that stores database diagnostic data such as trace files, the alert log, DDL log, and Health Monitor reports.
- Alert Log
  Every database has an **alert log**, which is a file containing a chronological log of database messages and errors.
- Attention Log
  Every database has an **attention log**, containing critical and highly visible database events.
- DDL Log
  The **DDL log** has the same format and basic behavior as the alert log but contains only DDL statements and details. The database writes DDL information to its own file to reduce the clutter in the alert log.
- Trace Files
  A **trace file** is a file that contains diagnostic data used to investigate problems. Also, trace files can provide guidance for tuning applications or an instance.
- Diagnostic Dumps
  A **diagnostic dump file** is a special type of trace file that contains detailed point-in-time information about a state or structure.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to manage diagnostic files

## Automatic Diagnostic Repository

**Automatic Diagnostic Repository (ADR)** is a file-based repository that stores database diagnostic data such as trace files, the alert log, DDL log, and Health Monitor reports.

Key characteristics of ADR include:

- Unified directory structure
- Consistent diagnostic data formats
- Unified tool set

The preceding characteristics enable customers and Oracle Support to correlate and analyze diagnostic data across multiple Oracle instances, components, and products.

ADR is located *outside* the database, which enables Oracle Database to access and manage ADR when the physical database is unavailable. A database instance can create ADR before a database has been created.

- Problems and Incidents
  ADR proactively tracks **problems**, which are critical errors in the database.

- ADR Structure
  The **ADR base** is the ADR root directory.

## Problems and Incidents

ADR proactively tracks **problems**, which are critical errors in the database.

Critical errors manifest as internal errors, such as `ORA-600`, or other severe errors. Each problem has a **problem key**, which is a text string that describes the problem.

When a problem occurs multiple times, ADR creates a time-stamped **incident** for each occurrence. An incident is uniquely identified by a numeric **incident ID**. When an incident occurs, ADR sends an **incident alert** to Enterprise Manager. Diagnosis and resolution of a critical error usually starts with an incident alert.

Because a problem could generate many incidents in a short time, ADR applies flood control to incident generation after certain thresholds are reached. A **flood-controlled incident** generates an alert log entry, but does not generate incident dumps. In this way, ADR informs you that a critical error is ongoing without overloading the system with diagnostic data.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for detailed information about the fault diagnosability infrastructure

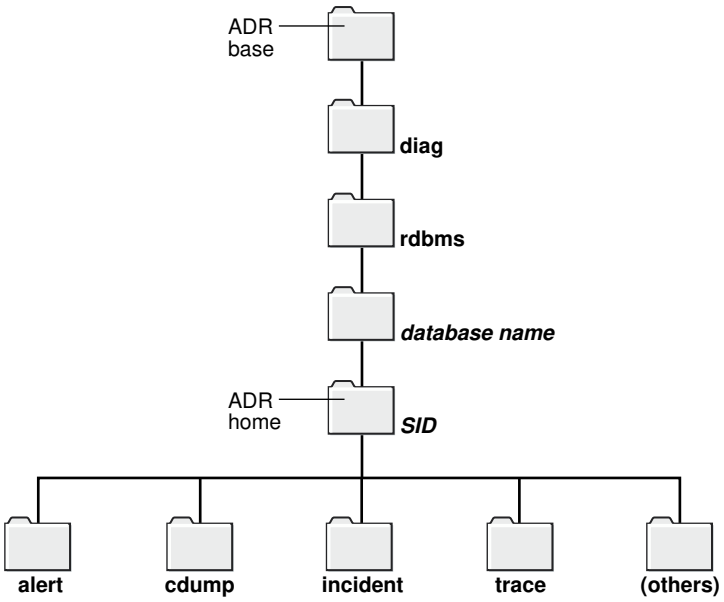
## ADR Structure

The **ADR base** is the ADR root directory.

The ADR base can contain multiple ADR homes, where each ADR home is the root directory for all diagnostic data—traces, dumps, the alert log, and so on—for an instance of an Oracle product or component. For example, in an Oracle RAC environment with shared storage and Oracle ASM, each database instance and each Oracle ASM instance has its own ADR home.

Figure 15-8 illustrates the ADR directory hierarchy for a database instance. Other ADR homes for other Oracle products or components, such as Oracle ASM or Oracle Net Services, can exist within this hierarchy, under the same ADR base.

**Figure 15-8 ADR Directory Structure for an Oracle Database Instance**




As the following Linux example shows, when you start an instance with a unique SID and database name *before* creating a database, Oracle Database creates ADR by default as a directory structure in the host file system. The SID and database name form part of the path name for files in the ADR Home.

**Example 15-1 Creation of ADR**

```
% setenv ORACLE_SID osi
% echo "DB_NAME=dbn" > init.ora
% sqlplus / as sysdba
.
.
.
Connected to an idle instance.

SQL> STARTUP NOMOUNT PFILE="./init.ora"
ORACLE instance started.

Total System Global Area  146472960 bytes
Fixed Size                  1317424 bytes
Variable Size              92276176 bytes
Database Buffers           50331648 bytes
Redo Buffers                2547712 bytes

SQL> COL NAME FORMAT a21
SQL> COL VALUE FORMAT a60
SQL> SELECT NAME, VALUE FROM V$DIAG_INFO;

NAME                  VALUE
--------------------- ------------------------------------------------------------
Diag Enabled          TRUE
ADR Base              /d1/3910926111/oracle/log
ADR Home              /d1/3910926111/oracle/log/diag/rdbms/dbn/osi
```

```
Diag Trace              /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/trace
Diag Alert              /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/alert
Diag Incident           /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/incident
Diag Cdump              /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/cdump
Health Monitor          /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/hm
Default Trace File      /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/trace/
osi_ora_29229.trc
Active Problem Count    0
Active Incident Count   0
ORACLE_HOME             /d1/3910926111/oracle
Attention Log           /d1/3910926111/oracle/log/diag/rdbms/dbn/osi/trace/
attention_osi.log
13 rows selected.
```

## Alert Log

Every database has an **alert log**, which is a file containing a chronological log of database messages and errors.

The alert log contents include the following:

- All internal errors (`ORA-600`), block corruption errors (`ORA-1578`), and deadlock errors (`ORA-60`)
- Administrative operations such as the SQL*Plus commands `STARTUP`, `SHUTDOWN`, `ARCHIVE LOG`, and `RECOVER`
- Several messages and errors relating to the functions of shared server and dispatcher processes
- Errors during the automatic refresh of a materialized view

If an administrative operation is successful, then Oracle Database writes a message to the alert log as "completed" along with a time stamp.

Oracle Database creates an alert log in the `alert` subdirectory shown in Figure 15-8 when you first start a database instance, even if no database has been created yet. This file is in the XML format. The `trace` subdirectory contains a text-only alert log.

Query `V$DIAG_INFO` to locate the alert log.

## Attention Log

Every database has an **attention log**, containing critical and highly visible database events.

The attention log includes only critical and highly visible database events, enabling you to identify an alert and take the appropriate action. Each attention log message has various attributes including target audience, cause, and an action to take. The attention log provides information in the following format:

- Attention message ID
- Attention message type: error, notification, warning
- Attention message urgency: DEFERABLE, IMMEDIATE, INFO, SOON
- Attention message scope: CDB, CDB cluster, CDB instance, PDB, session, PDB instance, process, session
- Attention message text

- Attention message target user: CDB Admin, Clusterware Admin, PDB Admin
- Cause
- Action

Oracle Database creates an attention log named `attention_ORACLE_SID.log` in the `trace` subdirectory shown in Figure 15-8. You can also query `V$DIAG_INFO` to locate the attention log.

## DDL Log

The **DDL log** has the same format and basic behavior as the alert log but contains only DDL statements and details. The database writes DDL information to its own file to reduce the clutter in the alert log.

DDL log records are DDL text, optionally augmented with supplemental information. One log record exists for each DDL statement. The DDL log is stored in the `log/ddl` subdirectory of the ADR home.

## Trace Files

A **trace file** is a file that contains diagnostic data used to investigate problems. Also, trace files can provide guidance for tuning applications or an instance.

- Types of Trace Files
  Each server and background process can periodically write to an associated trace file. The files contain information on the process environment, status, activities, and errors.
- Locations of Trace Files
  ADR stores trace files in the `trace` subdirectory. Trace file names are platform-dependent and use the extension `.trc`.
- Segmentation of Trace Files
  When the trace file size is limited, the database may automatically split it into a maximum of five segments. Segments are separate files that have the same name as the active trace file, but with a segment number appended, as in `ora_1234_2.trc`.

> **See Also:**
>
> *Oracle Database 2 Day + Performance Tuning Guide*

### Types of Trace Files

Each server and background process can periodically write to an associated trace file. The files contain information on the process environment, status, activities, and errors.

The SQL trace facility also creates trace files, which provide performance information on individual SQL statements. You can enable tracing for a client identifier, service, module, action, session, instance, or database in various ways. For example, you can execute the appropriate procedures in the `DBMS_MONITOR` package or set events.

> **See Also:**
>
> - "Session Control Statements"
> - *Oracle Database Administrator's Guide* to learn about trace files, dumps, and core files
> - *Oracle Database SQL Tuning Guide* to learn about application tracing

## Locations of Trace Files

ADR stores trace files in the `trace` subdirectory. Trace file names are platform-dependent and use the extension `.trc`.

Typically, database background process trace file names contain the Oracle SID, the background process name, and the operating system process number. An example of a trace file for the `RECO` process is `mytest_reco_10355.trc`.

Server process trace file names contain the Oracle SID, the string `ora`, and the operating system process number. An example of a server process trace file name is `mytest_ora_10304.trc`.

Sometimes trace files have corresponding trace metadata files, which end with the extension `.trm`. These files contain structural information called **trace maps** that the database uses for searching and navigation.

> **See Also:**
>
> - "Figure 15-8"
> - *Oracle Database Administrator's Guide* to learn how to find trace files

## Segmentation of Trace Files

When the trace file size is limited, the database may automatically split it into a maximum of five segments. Segments are separate files that have the same name as the active trace file, but with a segment number appended, as in `ora_1234_2.trc`.

Each segment is typically 20% of the limit set by `MAX_DUMP_FILE_SIZE`. When the combined size of all segments exceeds the limit, the database deletes the oldest segment (although never the first segment, which may contain relevant information about the initial state of the process), and then creates a new, empty segment.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to control the size of trace files

# Diagnostic Dumps

A **diagnostic dump file** is a special type of trace file that contains detailed point-in-time information about a state or structure.

A trace tends to be continuous output of diagnostic data. In contrast, a dump is typically a one-time output of diagnostic data in response to an event.

- Trace Dumps and Incidents
  Most dumps occur because of incidents.

## Trace Dumps and Incidents

Most dumps occur because of incidents.

When an incident occurs, the database writes one or more dumps to the incident directory created for the incident. Incident dumps also contain the incident number in the file name.

During incident creation, an application may take a heap or system state dump as part of an action. In such cases, the database appends the dump name to the incident file name instead of the default trace file name. For example, because of an incident in a process, the database creates file `prod_ora_90348.trc`. A dump in the incident generates the file `prod_ora_90348_`*`incident_id`*`.trc`, where *`incident_id`* is the numeric ID of the incident. A heap dump action created as part of the incident generates the heap dump file `prod_ora_90348_`*`incident_id_dump_id`*`.trc`, where *`dump_id`* is the numeric ID of the trace dump.

16

# Memory Architecture

This chapter discusses the memory architecture of a database instance.

This chapter contains the following sections:


- Introduction to Oracle Database Memory Structures
- Overview of the User Global Area
- Overview of the Program Global Area (PGA)
- Overview of the System Global Area (SGA)
- Overview of Software Code Areas


- Introduction to Oracle Database Memory Structures
  When an instance is started, Oracle Database allocates a memory area and starts background processes.
- Overview of the User Global Area
  The UGA is session memory, which is memory allocated for session variables, such as logon information, and other information required by a database session. Essentially, the UGA stores the session state.
- Overview of the Program Global Area (PGA)
  The PGA is memory specific to an operating process or thread that is not shared by other processes or threads on the system. Because the PGA is process-specific, it is never allocated in the SGA.
- Overview of the System Global Area (SGA)
  The **SGA** is a read/write memory area that, along with the Oracle background processes, form a database instance.
- Overview of Software Code Areas
  A **software code area** is a portion of memory that stores code that is being run or can be run. Oracle Database code is stored in a software area that is typically more exclusive and protected than the location of user programs.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for instructions for configuring and managing memory

## Introduction to Oracle Database Memory Structures

When an instance is started, Oracle Database allocates a memory area and starts background processes.

The memory area stores information such as the following:

- Program code

- Information about each connected session, even if it is not currently active
- Information needed during program execution, for example, the current state of a query from which rows are being fetched
- Information such as lock data that is shared and communicated among processes
- Cached data, such as data blocks and redo records, that also exists on disk

- Basic Memory Structures
  Oracle Database includes several memory areas, each of which contains multiple subcomponents.
- Oracle Database Memory Management
  Memory management involves maintaining optimal sizes for the Oracle instance memory structures as demands on the database change. Oracle Database manages memory based on the settings of memory-related initialization parameters.

> **See Also:**
>
> "Process Architecture"

## Basic Memory Structures

Oracle Database includes several memory areas, each of which contains multiple subcomponents.

The basic memory structures associated with Oracle Database include:

- System global area (SGA)

  The SGA is a group of shared memory structures, known as *SGA components*, that contain data and control information for one Oracle Database instance. All server and background processes share the SGA. Examples of data stored in the SGA include cached data blocks and shared SQL areas.
- Program global area (PGA)

  A PGA is a nonshared memory region that contains data and control information exclusively for use by an Oracle process. Oracle Database creates the PGA when an Oracle process starts.

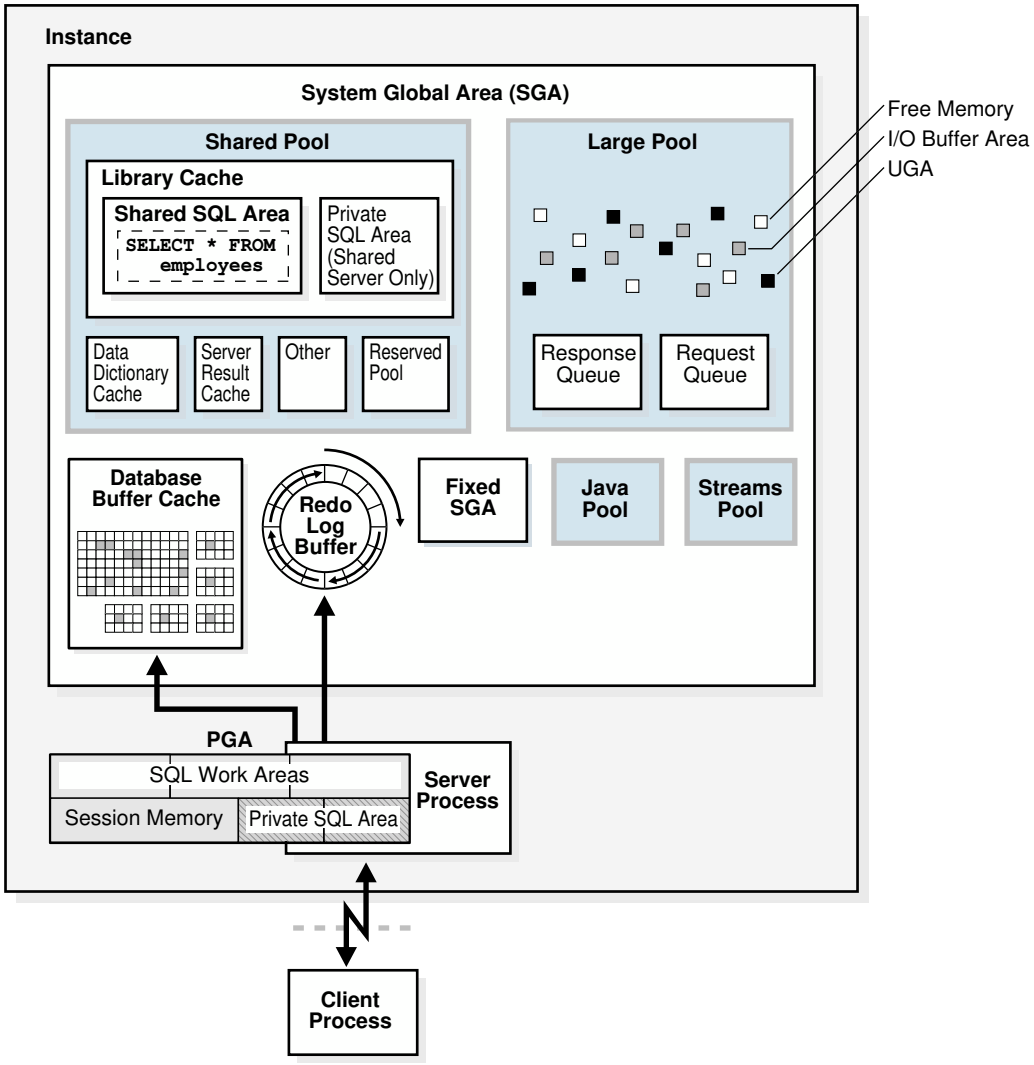
  One PGA exists for each server process and background process. The collection of individual PGAs is the total instance PGA, or instance PGA. Database initialization parameters set the size of the instance PGA, not individual PGAs.
- User global area (UGA)

  The UGA is memory associated with a user session.
- Software code areas

  Software code areas are portions of memory used to store code that is being run or can be run. Oracle Database code is stored in a software area that is typically at a different location from user programs—a more exclusive or protected location.

The following figure illustrates the relationships among these memory structures.

**Figure 16-1 Oracle Database Memory Structures**




# Oracle Database Memory Management

Memory management involves maintaining optimal sizes for the Oracle instance memory structures as demands on the database change. Oracle Database manages memory based on the settings of memory-related initialization parameters.

The basic options for memory management are as follows:

- Automatic memory management

  You specify the target size for the database instance memory. The instance automatically tunes to the target memory size, redistributing memory as needed between the SGA and the instance PGA.

- Automatic shared memory management

  This management mode is partially automated. You set a target size for the SGA and then have the option of setting an aggregate target size for the PGA or managing PGA work areas individually.

- Manual memory management

Instead of setting the total memory size, you set many initialization parameters to manage components of the SGA and instance PGA individually.

If you create a database with Database Configuration Assistant (DBCA) and choose the basic installation option, then automatic memory management is the default.

**See Also:**

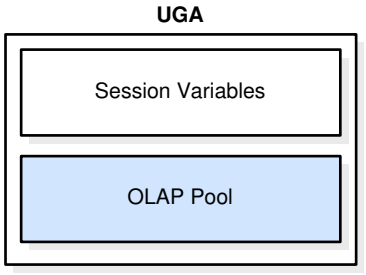
- *Oracle Database Performance Tuning Guide* for more information about memory management options for DBAs
- *Oracle Database Administrator's Guide* to learn about memory management options

# Overview of the User Global Area

The UGA is session memory, which is memory allocated for session variables, such as logon information, and other information required by a database session. Essentially, the UGA stores the session state.

The following figure depicts the UGA.

**Figure 16-2 User Global Area (UGA)**




If a session loads a PL/SQL package into memory, then the UGA contains the *package state*, which is the set of values stored in all the package variables at a specific time. The package state changes when a package subprogram changes the variables. By default, the package variables are unique to and persist for the life of the session.

The OLAP page pool is also stored in the UGA. This pool manages OLAP data pages, which are equivalent to data blocks. The page pool is allocated at the start of an OLAP session and released at the end of the session. An OLAP session opens automatically whenever a user queries a dimensional object such as a cube.

The UGA must be available to a database session for the life of the session. For this reason, the UGA cannot be stored in the PGA when using a shared server connection because the PGA is specific to a single process. Therefore, the UGA is stored in the SGA when using shared server connections, enabling any shared server process access to it. When using a dedicated server connection, the UGA is stored in the PGA.

> **See Also:**
>
> - "PL/SQL Packages"
> - "Connections and Sessions"
> - *Oracle Database Net Services Administrator's Guide* to learn about shared server connections
> - *Oracle OLAP User's Guide* for an overview of Oracle OLAP

# Overview of the Program Global Area (PGA)

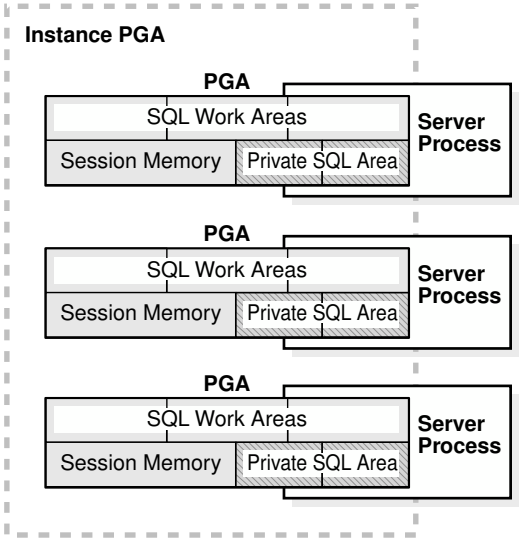
The PGA is memory specific to an operating process or thread that is not shared by other processes or threads on the system. Because the PGA is process-specific, it is never allocated in the SGA.

The PGA is a memory heap that contains session-dependent variables required by a dedicated or shared server process. The server process allocates memory structures that it requires in the PGA.

An analogy for a PGA is a temporary countertop workspace used by a file clerk. In this analogy, the file clerk is the server process doing work on behalf of the customer (client process). The clerk clears a section of the countertop, uses the workspace to store details about the customer request and to sort the folders requested by the customer, and then gives up the space when the work is done.

The following figure shows an instance PGA (collection of all PGAs) for an instance that is not configured for shared servers. You can use an initialization parameter to set a target maximum size of the instance PGA. Individual PGAs can grow as needed up to this target size.

**Figure 16-3 Instance PGA**



> **Note:**
>
> Background processes also allocate their own PGAs. This discussion focuses on server process PGAs only.

- Contents of the PGA
  The PGA is subdivided into different areas, each with a different purpose.
- PGA Usage in Dedicated and Shared Server Modes
  PGA memory allocation depends on whether the database uses dedicated or shared server connections.

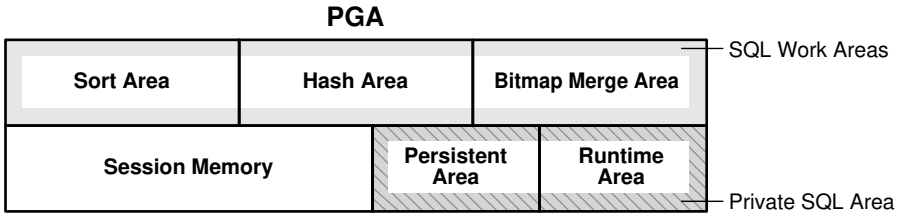
> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for an overview of PGA memory management

## Contents of the PGA

The PGA is subdivided into different areas, each with a different purpose.

The following figure shows the possible contents of the PGA for a dedicated server session. Not all of the PGA areas will exist in every case.

**Figure 16-4 PGA Contents**




- Private SQL Area
  A **private SQL area** holds information about a parsed SQL statement and other session-specific information for processing.
- SQL Work Areas
  A **work area** is a private allocation of PGA memory used for memory-intensive operations.

## Private SQL Area

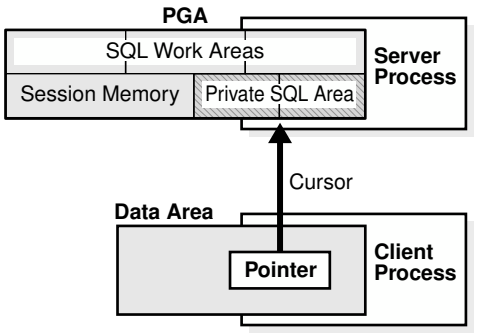
A **private SQL area** holds information about a parsed SQL statement and other session-specific information for processing.

When a server process executes SQL or PL/SQL code, the process uses the private SQL area to store bind variable values, query execution state information, and query execution work areas.

Do not confuse a *private* SQL area, which is in the PGA, with the *shared* SQL area, which stores execution plans in the SGA. Multiple private SQL areas in the same or different sessions can point to a single execution plan in the SGA. For example, 20 executions of `SELECT * FROM sales` in one session and 10 executions of the same query in a different session can share the same plan. The private SQL areas for each execution are not shared and may contain different values and data.

A cursor is a name or handle to a specific private SQL area. As shown in the following graphic, you can think of a cursor as a pointer on the client side and as a state on the server side. Because cursors are closely associated with private SQL areas, the terms are sometimes used interchangeably.

**Figure 16-5 Cursor**




A private SQL area is divided into the following areas:

- The run-time area

  This area contains query execution state information. For example, the run-time area tracks the number of rows retrieved so far in a full table scan.

  Oracle Database creates the run-time area as the first step of an execute request. For DML statements, the run-time area is freed when the SQL statement is closed.

- The persistent area

  This area contains bind variable values. A bind variable value is supplied to a SQL statement at run time when the statement is executed. The persistent area is freed only when the cursor is closed.

The client process is responsible for managing private SQL areas. The allocation and deallocation of private SQL areas depends largely on the application, although the number of private SQL areas that a client process can allocate is limited by the initialization parameter `OPEN_CURSORS`.

Although most users rely on the automatic cursor handling of database utilities, the Oracle Database programmatic interfaces offer developers more control over cursors. In general, applications should close all open cursors that will not be used again to free the persistent area and to minimize the memory required for application users.

> **See Also:**
>
> - "Shared SQL Areas"
> - *Oracle Database Development Guide* and *Oracle Database PL/SQL Language Reference* to learn how to use cursors

## SQL Work Areas

A **work area** is a private allocation of PGA memory used for memory-intensive operations.

For example, a sort operator uses the sort area to sort a set of rows. Similarly, a hash join operator uses a hash area to build a hash table from its left input, whereas a bitmap merge uses the bitmap merge area to merge data retrieved from scans of multiple bitmap indexes.

The following example shows a join of `employees` and `departments` with its query plan:

```
SQL> SELECT *
  2  FROM   employees e JOIN departments d
  3  ON     e.department_id=d.department_id
  4  ORDER BY last_name;
.
.
.
------------------------------------------------------------------------------
--
| Id| Operation           | Name        | Rows  | Bytes | Cost (%CPU)|
Time     |
------------------------------------------------------------------------------
--
| 0 | SELECT STATEMENT    |             |   106 |  9328 |    7  (29)|
00:00:01 |
| 1 |  SORT ORDER BY      |             |   106 |  9328 |    7  (29)|
00:00:01 |
|*2 |   HASH JOIN         |             |   106 |  9328 |    6  (17)|
00:00:01 |
| 3 |    TABLE ACCESS FULL| DEPARTMENTS |    27 |   540 |    2   (0)|
00:00:01 |
| 4 |    TABLE ACCESS FULL| EMPLOYEES   |   107 |  7276 |    3   (0)|
00:00:01 |
------------------------------------------------------------------------------
--
```

In the preceding example, the run-time area tracks the progress of the full table scans. The session performs a hash join in the hash area to match rows from the two tables. The `ORDER BY` sort occurs in the sort area.

If the amount of data to be processed by the operators does not fit into a work area, then Oracle Database divides the input data into smaller pieces. In this way, the database processes some data pieces in memory while writing the rest to temporary disk storage for processing later.

The database automatically tunes work area sizes when automatic PGA memory management is enabled. You can also manually control and tune the size of a work area.

Generally, larger work areas can significantly improve performance of an operator at the cost of higher memory consumption. Optimally, the size of a work area is sufficient to accommodate the input data and auxiliary memory structures allocated by its associated SQL operator. If not, response time increases because part of the input data must be cached on disk. In the extreme case, if the size of a work area is too small compared to input data size, then the database must perform multiple passes over the data pieces, dramatically increasing response time.

**See Also:**

- *Oracle Database Administrator's Guide* to learn how to use automatic PGA management
- *Oracle Database Performance Tuning Guide* to learn how to tune PGA memory

## PGA Usage in Dedicated and Shared Server Modes

PGA memory allocation depends on whether the database uses dedicated or shared server connections.

The following table shows the differences.

**Table 16-1 Differences in Memory Allocation Between Dedicated and Shared Servers**

| Memory Area | Dedicated Server | Shared Server |
|---|---|---|
| Nature of session memory | Private | Shared |
| Location of the persistent area | PGA | SGA |
| Location of the run-time area for DML and DDL statements | PGA | PGA |

**See Also:**

*Oracle Database Administrator's Guide* to learn how to configure a database for shared server

# Overview of the System Global Area (SGA)

The **SGA** is a read/write memory area that, along with the Oracle background processes, form a database instance.

**Note:**

The server and background processes do not reside *within* the SGA, but exist in a separate memory space.

All server processes that execute on behalf of users can read information in the instance SGA. Several processes write to the SGA during database operation.

Each database instance has its own SGA. Oracle Database automatically allocates memory for an SGA at instance startup and reclaims the memory at instance shutdown. When you start an instance with SQL*Plus or Oracle Enterprise Manager, the size of the SGA is shown as in the following example:

```
SQL> STARTUP
ORACLE instance started.

Total System Global Area  368283648 bytes
Fixed Size                  1300440 bytes
Variable Size             343935016 bytes
Database Buffers           16777216 bytes
Redo Buffers                6270976 bytes
Database mounted.
Database opened.
```

As shown in Figure 16-1, the SGA consists of several memory components, which are pools of memory used to satisfy a particular class of memory allocation requests. All SGA components except the redo log buffer allocate and deallocate space in units of contiguous memory called *granules*. Granule size is platform-specific and is determined by total SGA size.

You can query the `V$SGASTAT` view for information about SGA components.

The most important SGA components are the following:

- Database Buffer Cache
- In-Memory Area
- Redo Log Buffer
- Shared Pool
- Large Pool
- Java Pool
- Fixed SGA
- Optional Performance-Related SGA Subareas

- Database Buffer Cache
  The **database buffer cache**, also called the *buffer cache*, is the memory area that stores copies of data blocks read from data files.
- Redo Log Buffer
  The **redo log buffer** is a circular buffer in the SGA that stores redo entries describing changes made to the database.
- Shared Pool
  The **shared pool** caches various types of program data.
- Large Pool
  The **large pool** is an optional memory area intended for memory allocations that are larger than is appropriate for the shared pool.

- Java Pool
  The **Java pool** is an area of memory that stores all session-specific Java code and data within the Java Virtual Machine (JVM). This memory includes Java objects that are migrated to the Java session space at end-of-call.
- Fixed SGA
  The **fixed SGA** is an internal housekeeping area.
- Optional Performance-Related SGA Subareas
  Some SGA subareas are only enabled for specific performance features.

> **See Also:**
>
> - "Introduction to the Oracle Database Instance"
> - *Oracle Database Performance Tuning Guide* to learn more about granule sizing

## Database Buffer Cache

The **database buffer cache**, also called the *buffer cache*, is the memory area that stores copies of data blocks read from data files.

A buffer is a main memory address in which the buffer manager temporarily caches a currently or recently used data block. All users concurrently connected to a database instance share access to the buffer cache.

- Purpose of the Database Buffer Cache
  Oracle Database uses the buffer cache to achieve multiple goals.
- Buffer States
  The database uses internal algorithms to manage buffers in the cache.
- Buffer Modes
  When a client requests data, Oracle Database retrieves buffers from the database buffer cache in either current mode or consistent mode.
- Buffer I/O
  A **logical I/O**, also known as a *buffer I/O*, refers to reads and writes of buffers in the buffer cache.
- Buffer Pools
  A **buffer pool** is a collection of buffers.
- Buffers and Full Table Scans
  The database uses a complicated algorithm to manage table scans. By default, when buffers must be read from disk, the database inserts the buffers into the middle of the LRU list. In this way, hot blocks can remain in the cache so that they do not need to be read from disk again.
- DRAM and PMEM Buffers
  Standard (DRAM) database buffers differ from PMEM buffers, but they share characteristics.

### Purpose of the Database Buffer Cache

Oracle Database uses the buffer cache to achieve multiple goals.

The goals include:

- Optimize physical I/O

  The database updates data blocks in the cache and stores metadata about the changes in the redo log buffer. After a `COMMIT`, the database writes the redo buffers to the online redo log but does not immediately write data blocks to the data files. Instead, database writer (DBW) performs lazy writes in the background.

- Keep frequently accessed blocks in the buffer cache and write infrequently accessed blocks to disk

  When Database Smart Flash Cache (flash cache) is enabled, part of the buffer cache can reside in the flash cache. This buffer cache extension is stored on one or more flash devices. The database can improve performance by caching buffers in flash memory instead of reading from magnetic disk.

  Use the `DB_FLASH_CACHE_FILE` and `DB_FLASH_CACHE_SIZE` initialization parameters to configure multiple flash devices. The buffer cache tracks each device and distributes buffers to the devices uniformly.

> **Note:**
>
> Database Smart Flash Cache is available only in Solaris and Oracle Linux.

- Manage buffer headers that point to data files in Oracle Persistent Memory Filestore (PMEM Filestore)

  If you enable PMEM Filestore, then database files are mapped for direct read-only access. Queries can bypass the traditional buffer cache mechanism, avoiding unnecessary I/O. In this case, buffer headers must store metadata corresponding to the PMEM blocks. The database can still use the traditional (DRAM) buffer cache for modifications, read consistency, and faster access for "hot" data blocks.

> **See Also:**
>
> - "Oracle Persistent Memory Filestore (PMEM Filestore)"
> - "DRAM and PMEM Buffers"
> - *Oracle Database Reference* to learn about the `DB_FLASH_CACHE_FILE` initialization parameter

## Buffer States

The database uses internal algorithms to manage buffers in the cache.

A buffer can be in any of the following mutually exclusive states:

- Unused

  The buffer is available for use because it has never been used or is currently unused. This type of buffer is the easiest for the database to use.

- Clean

This buffer was used earlier and now contains a read-consistent version of a block as of a point in time. The block contains data but is "clean" so it does not need to be checkpointed. The database can pin the block and reuse it.

- Dirty

  The buffer contain modified data that has not yet been written to disk. The database must checkpoint the block before reusing it.

Every buffer has an access mode: pinned or free (unpinned). A buffer is "pinned" in the cache so that it does not age out of memory while a user session accesses it. Multiple sessions cannot modify a pinned buffer at the same time.

## Buffer Modes

When a client requests data, Oracle Database retrieves buffers from the database buffer cache in either current mode or consistent mode.

The modes differ as follows:

- Current mode

  A current mode get, also called a *db block get*, is a retrieval of a block as it currently appears in the buffer cache. For example, if an uncommitted transaction has updated two rows in a block, then a current mode get retrieves the block with these uncommitted rows. The database uses db block gets most frequently during modification statements, which must update only the current version of the block.

- Consistent mode

  A consistent read get is a retrieval of a read-consistent version of a block. This retrieval may use undo data. For example, if an uncommitted transaction has updated two rows in a block, and if a query in a separate session requests the block, then the database uses undo data to create a read-consistent version of this block (called a *consistent read clone*) that does not include the uncommitted updates. Typically, a query retrieves blocks in consistent mode.

> **See Also:**
>
> - "Read Consistency and Undo Segments"
> - *Oracle Database Reference* for descriptions of database statistics such as db block get and consistent read get

## Buffer I/O

A **logical I/O**, also known as a *buffer I/O*, refers to reads and writes of buffers in the buffer cache.

When a requested buffer is not found in memory, the database performs a physical I/O to copy the buffer from either the flash cache or disk into memory. The database then performs a logical I/O to read the cached buffer.

- Buffer Replacement Algorithms
  To make buffer access efficient, the database must decide which buffers to cache in memory, and which to access from disk.

- Buffer Writes
  The **database writer (DBW)** process periodically writes cold, dirty buffers to disk.
- Buffer Reads
  When the number of unused buffers is low, the database must remove buffers from the buffer cache.
- Buffer Touch Counts
  The database measures the frequency of access of buffers on the LRU list using a touch count. This mechanism enables the database to increment a counter when a buffer is pinned instead of constantly shuffling buffers on the LRU list.

## Buffer Replacement Algorithms

To make buffer access efficient, the database must decide which buffers to cache in memory, and which to access from disk.

The database uses the following algorithms:

- LRU-based, block-level replacement algorithm

  This sophisticated algorithm, which is the default, uses a least recently used (LRU) list that contains pointers to dirty and non-dirty buffers. The LRU list has a hot end and cold end. A cold buffer is a buffer that has not been recently used. A hot buffer is frequently accessed and has been recently used. Conceptually, there is only one LRU, but for data concurrency the database actually uses several LRUs.
- Temperature-based, object-level replacement algorithm

  Starting in Oracle Database 12c Release 1 (12.1.0.2), the automatic big table caching feature enables table scans to use a different algorithm in the following scenarios:

  – Parallel queries

    In single-instance and Oracle Real Applications Cluster (Oracle RAC) databases, parallel queries can use the big table cache when the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter is set to a nonzero value, and `PARALLEL_DEGREE_POLICY` is set to `auto` or `adaptive`.

  – Serial queries

    In a single-instance configuration only, serial queries can use the big table cache when the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter is set to a nonzero value.

  When a table does not fit in memory, the database decides which buffers to cache based on access patterns. For example, if only 95% of a popular table fits in memory, then the database may choose to leave 5% of the blocks on disk rather than cyclically reading blocks into memory and writing blocks to disk—a phenomenon known as *thrashing*. When caching multiple large objects, the database considers more popular tables hotter and less popular tables cooler, which influences which blocks are cached. The `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter sets the percentage of the buffer cache that uses this algorithm.

> **Note:**
>
> This document explains the LRU-based, block level replacement algorithm.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* to learn more about the temperature-based algorithm

## Buffer Writes

The **database writer (DBW)** process periodically writes cold, dirty buffers to disk.

DBW writes buffers in the following circumstances:

- A server process cannot find clean buffers for reading new blocks into the database buffer cache.

  As buffers are dirtied, the number of free buffers decreases. If the number drops below an internal threshold, and if clean buffers are required, then server processes signal DBW to write.

  The database uses the LRU to determine which dirty buffers to write. When dirty buffers reach the cold end of the LRU, the database moves them off the LRU to a write queue. DBW writes buffers in the queue to disk, using multiblock writes if possible. This mechanism prevents the end of the LRU from becoming clogged with dirty buffers and allows clean buffers to be found for reuse.
- The database must advance the checkpoint, which is the position in the redo thread from which instance recovery must begin.
- Tablespaces are changed to read-only status or taken offline.

> **See Also:**
>
> - "Database Writer Process (DBW)"
> - *Oracle Database Performance Tuning Guide* to learn how to diagnose and tune buffer write issues

## Buffer Reads

When the number of unused buffers is low, the database must remove buffers from the buffer cache.

The algorithm depends on whether the flash cache is enabled:

- Flash cache disabled

  The database re-uses each clean buffer as needed, overwriting it. If the overwritten buffer is needed later, then the database must read it from magnetic disk.
- Flash cache enabled

  DBW can write the body of a clean buffer to the flash cache, enabling reuse of its in-memory buffer. The database keeps the buffer header in an LRU list in main memory to track the state and location of the buffer body in the flash cache. If this buffer is needed later, then the database can read it from the flash cache instead of from magnetic disk.

When a client process requests a buffer, the server process searches the buffer cache for the buffer. A cache hit occurs if the database finds the buffer in memory. The search order is as follows:

1. The server process searches for the whole buffer in the buffer cache.

   If the process finds the whole buffer, then the database performs a logical read of this buffer.

2. The server process searches for the buffer header in the flash cache LRU list.

   If the process finds the buffer header, then the database performs an optimized physical read of the buffer body from the flash cache into the in-memory cache.

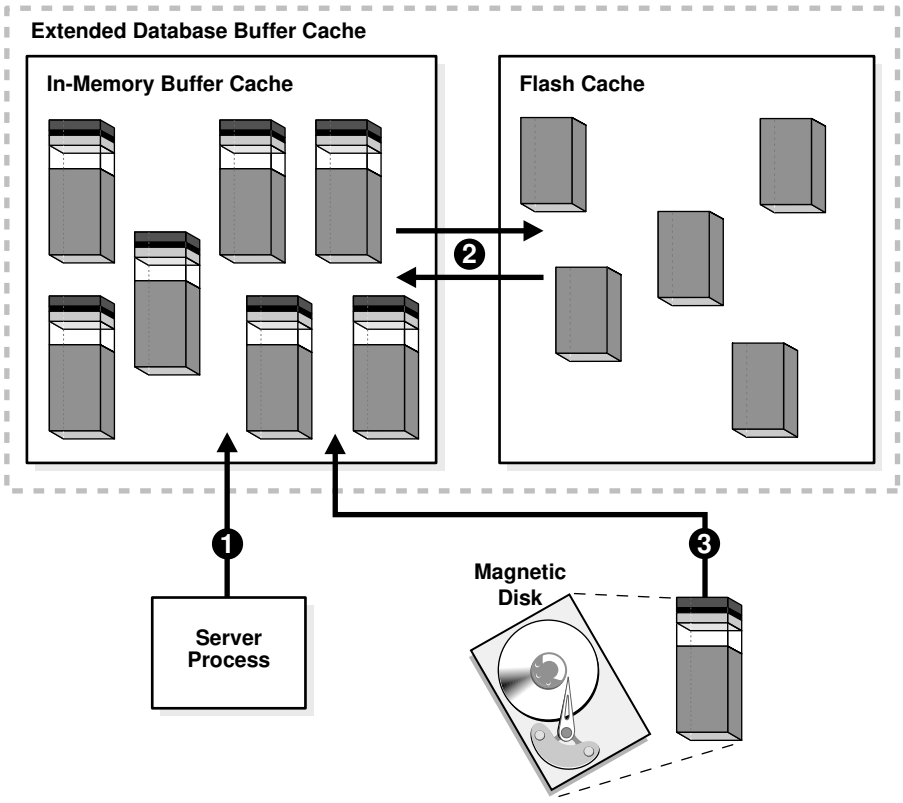
3. If the process does *not* find the buffer in memory (a cache miss), then the server process performs the following steps:

   a. Copies the block from a data file on disk into memory (a physical read)

   b. Performs a logical read of the buffer that was read into memory

Figure 16-6 illustrates the buffer search order. The extended buffer cache includes both the in-memory buffer cache, which contains whole buffers, and the flash cache, which contains buffer bodies. In the figure, the database searches for a buffer in the buffer cache and, not finding the buffer, reads it into memory from magnetic disk.

**Figure 16-6 Buffer Search**




In general, accessing data through a cache hit is faster than through a cache miss. The buffer cache hit ratio measures how often the database found a requested block in the buffer cache without needing to read it from disk.

The database can perform physical reads from either a data file or a temp file. Reads from a data file are followed by logical I/Os. Reads from a temp file occur when insufficient memory forces the database write data to a temporary table and read it back later. These physical reads bypass the buffer cache and do not incur a logical I/O.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* to learn how to calculate the buffer cache hit ratio

### Buffer Touch Counts

The database measures the frequency of access of buffers on the LRU list using a touch count. This mechanism enables the database to increment a counter when a buffer is pinned instead of constantly shuffling buffers on the LRU list.

> **Note:**
>
> The database does not physically move blocks in memory. The movement is the change in location of a pointer on a list.

When a buffer is pinned, the database determines when its touch count was last incremented. If the count was incremented over three seconds ago, then the count is incremented; otherwise, the count stays the same. The three-second rule prevents a burst of pins on a buffer counting as many touches. For example, a session may insert several rows in a data block, but the database considers these inserts as one touch.

If a buffer is on the cold end of the LRU, but its touch count is high, then the buffer moves to the hot end. If the touch count is low, then the buffer ages out of the cache.

## Buffer Pools

A **buffer pool** is a collection of buffers.

The database buffer cache is divided into one or more buffer pools, which manage blocks in mostly the same way. The pools do not have radically different algorithms for aging or caching blocks.

You can manually configure separate buffer pools that either keep data in the buffer cache or make the buffers available for new data immediately after using the data blocks. You can then assign specific schema objects to the appropriate buffer pool to control how blocks age out of the cache. For example, you can segregate segments into hot, warm, and cold buffer pools.

The possible buffer pools are as follows:

- Default pool

  This pool is the location where blocks are normally cached. Unless you manually configure separate pools, the default pool is the only buffer pool. The optional configuration of the other pools has no effect on the default pool.

The big table cache is an optional section of the default pool that uses a temperature-based, object-level replacement algorithm. In single-instance and Oracle RAC databases, parallel queries can use the big table cache when the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter is set to a nonzero value, and `PARALLEL_DEGREE_POLICY` is set to `auto` or `adaptive`. In single-instance configurations only, serial queries can use the big table cache when `DB_BIG_TABLE_CACHE_PERCENT_TARGET` is set.

- Keep pool

  This pool is intended for blocks that were accessed frequently, but which aged out of the default pool because of lack of space. The purpose of the keep buffer pool is to retain objects in memory, thus avoiding I/O operations.

  > **Note:**
  >
  > The keep pool manages buffers in the same way as the other pools: it does not use a special algorithm to pin buffers. The word "keep" is a naming convention. You can place tables that you want to keep in the larger keep pool, and place tables that you do not want to keep in the smaller recycle pool.

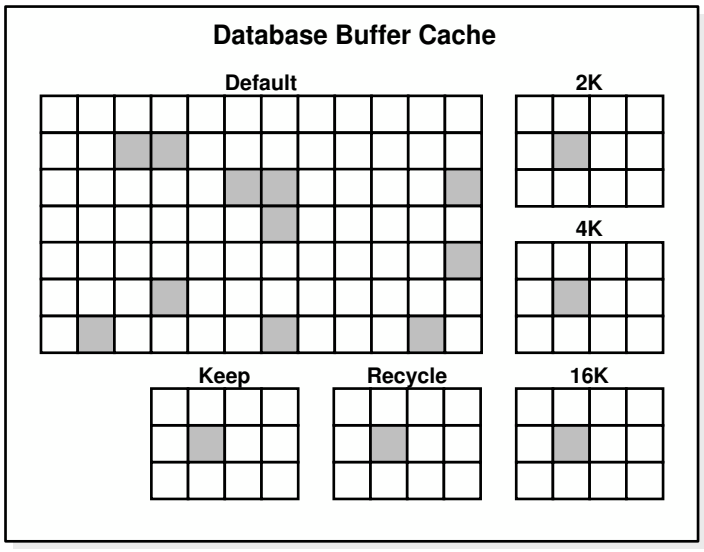
- Recycle pool

  This pool is intended for blocks that are used infrequently. A recycle pool prevent objects from consuming unnecessary space in the cache.

A database has a standard block size. You can create a tablespace with a block size that differs from the standard size. Each nondefault block size has its own pool. Oracle Database manages the blocks in these pools in the same way as in the default pool.

The following figure shows the structure of the buffer cache when multiple pools are used. The cache contains default, keep, and recycle pools. The default block size is 8 KB. The cache contains separate pools for tablespaces that use the nonstandard block sizes of 2 KB, 4 KB, and 16 KB.

**Figure 16-7 Database Buffer Cache**



> **See Also:**
>
> - "Database Block Size"
> - *Oracle Database Administrator's Guide* to learn more about buffer pools
> - *Oracle Database Performance Tuning Guide* to learn how to use multiple buffer pools
> - *Oracle Database Reference* to learn about the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter

## Buffers and Full Table Scans

The database uses a complicated algorithm to manage table scans. By default, when buffers must be read from disk, the database inserts the buffers into the middle of the LRU list. In this way, hot blocks can remain in the cache so that they do not need to be read from disk again.

A problem is posed by a full table scan, which sequentially reads all rows under the table high water mark (HWM). Suppose that the total size of the blocks in a table segment is greater than the size of the buffer cache. A full scan of this table could clean out the buffer cache, preventing the database from maintaining a cache of frequently accessed blocks.

- Default Mode for Full Table Scans
  By default, the database takes a conservative approach to full table scans, loading a small table into memory only when the table size is a small percentage of the buffer cache.
- Parallel Query Execution
  When performing a full table scan, the database can sometimes improve response time by using multiple parallel execution servers.
- CACHE Attribute
  In the rare case where the default caching behavior is not desired, you can use `ALTER TABLE ... CACHE` to change how blocks from large tables are read into the database buffer cache.
- KEEP Attribute
  For large tables, you can use `ALTER TABLE ... STORAGE BUFFER_POOL KEEP` to cause scans to load blocks for these tables into the keep pool.
- Force Full Database Caching Mode
  To improve performance in some situations, you can explicitly execute the `ALTER DATABASE ... FORCE FULL DATABASE CACHING` statement to enable the **force full database caching mode**.

> **See Also:**
>
> "Segment Space and the High Water Mark"

### Default Mode for Full Table Scans

By default, the database takes a conservative approach to full table scans, loading a small table into memory only when the table size is a small percentage of the buffer cache.

To determine whether medium sized tables should be cached, the database uses an algorithm that incorporates the interval between the last table scan, the aging timestamp of the buffer cache, and the space remaining in the buffer cache.

For very large tables, the database typically uses a direct path read, which loads blocks directly into the PGA and bypasses the SGA altogether, to avoid populating the buffer cache. For medium size tables, the database may use a direct read or a cache read. If it decides to use a cache read, then the database places the blocks at the end of the LRU list to prevent the scan from effectively cleaning out the buffer cache.

Starting in Oracle Database 12c Release 1 (12.1.0.2), the buffer cache of a database instance automatically performs an internal calculation to determine whether memory is sufficient for the database to be fully cached in the instance SGA, and if caching tables on access would be beneficial for performance. If the whole database can fully fit in memory, and if various other internal criteria are met, then Oracle Database treats all tables in the database as small tables, and considers them eligible for caching. However, the database does not cache LOBs marked with the `NOCACHE` attribute.

## Parallel Query Execution

When performing a full table scan, the database can sometimes improve response time by using multiple parallel execution servers.

In some cases, as when the database has a large amount of memory, the database can cache parallel query data in the system global area (SGA) instead of using direct path reads into the program global area (PGA). Typically, parallel queries occur in low-concurrency data warehouses because of the potential resource usage.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for an introduction to data warehouses
> - *Oracle Database VLDB and Partitioning Guide* to learn more about parallel execution

## CACHE Attribute

In the rare case where the default caching behavior is not desired, you can use `ALTER TABLE ... CACHE` to change how blocks from large tables are read into the database buffer cache.

For tables with the `CACHE` attribute set, the database does not force or pin the blocks in the buffer cache. Instead, the database ages the blocks out of the cache in the same way as any other table block. Use care when exercising this option because a full scan of a large table may clean most of the other blocks out of the cache.

> **Note:**
>
> Executing `ALTER TABLE ... CACHE` does not *cause* a table to be cached.

## KEEP Attribute

For large tables, you can use `ALTER TABLE ... STORAGE BUFFER_POOL KEEP` to cause scans to load blocks for these tables into the keep pool.

Placing a table into the keep pool changes the part of the buffer cache where the blocks are stored. Instead of caching blocks in the default buffer pool, the database caches them in the keep buffer pool. No separate algorithm controls keep pool caching.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the `CACHE` clause and the `KEEP` attribute
> - *Oracle Database Performance Tuning Guide* to learn how to interpret buffer cache advisory statistics

## Force Full Database Caching Mode

To improve performance in some situations, you can explicitly execute the `ALTER DATABASE ... FORCE FULL DATABASE CACHING` statement to enable the **force full database caching mode**.

In contrast to the default mode, which is automatic, the force full database caching mode considers the entire database, including `NOCACHE` LOBs, as eligible for caching in the database buffer cache. This mode is available starting in Oracle Database 12c Release 1 (12.1.0.2).

> **Note:**
>
> Enabling force full database caching mode does *not* force the database into memory. Rather, the entire database is *eligible* to be cached in the buffer cache. Oracle Database caches tables only when they are accessed.

Oracle recommends that you enable force full database caching mode only when the buffer cache size of each individual instance is greater than the database size. This guideline applies to both single-instance and Oracle RAC databases. However, when Oracle RAC applications are well partitioned, you can enable force full database caching mode when the combined buffer cache of all instances, with extra space to handle duplicate cached blocks between instances, is greater than the database size.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn how to enable force full database caching mode
> - *Oracle Database SQL Language Reference* for more information about `ALTER DATABASE ... FORCE FULL DATABASE CACHING` statement

## DRAM and PMEM Buffers

Standard (DRAM) database buffers differ from PMEM buffers, but they share characteristics.

When Oracle Persistent Memory Filestore (PMEM Filestore) is configured, every data block is directly mapped to the buffer cache in DRAM. Unlike DRAM buffers, PMEM buffers do *not* copy the contents of data blocks. Rather, PMEM buffer headers point to data blocks stored in PMEM Filestore. For most reads, the database only caches the block metadata, not the contents. PMEM buffers use a special type of granule whose structure is different from standard buffer cache granules.

> **Note:**
>
> The initialization parameter `DB_CACHE_SIZE` specifies the minimum size of the DRAM cache. PMEM metadata overhead is not included in this allocation.

PMEM buffers can be in any of the following states:

- Current

  This is the current version of a PMEM buffer. It can be directly accessed from the filestore.

  Oracle Database keeps the PMEM current version separate from the standard DRAM buffer current version. This separation helps to reduce code complexity during PMEM block pinning, cleanup, and migrate to DRAM.

- Consistent

  This is the consistent read version of a PMEM buffer. It is created after the database creates a clone in DRAM. The PMEM buffer can be directly accessed from the filestore.

- Free

  This is a free PMEM buffer. It can be reused by a PMEM block. After instance startup, all PMEM buffers are in a free state.

PMEM has higher latency than DRAM. Oracle Database uses an internal, workload-based algorithm to decide which blocks to migrate from PMEM to DRAM.

> **See Also:**
>
> - "Oracle Persistent Memory Filestore (PMEM Filestore)"
> - *Oracle Database Administrator's Guide* to learn how to set up PMEM Filestore

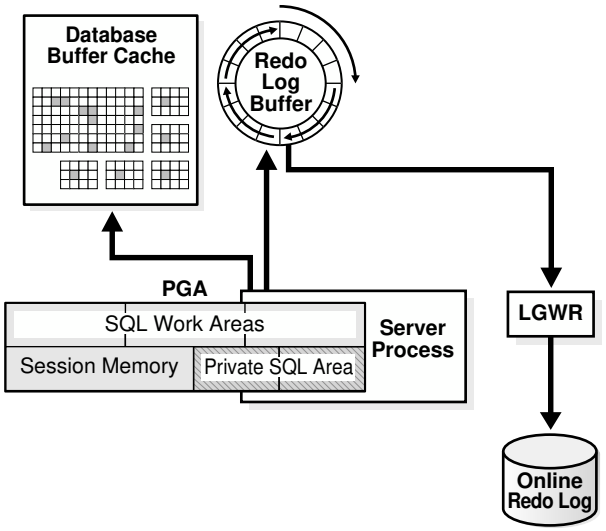
# Redo Log Buffer

The **redo log buffer** is a circular buffer in the SGA that stores redo entries describing changes made to the database.

A redo record is a data structure that contains the information necessary to reconstruct, or redo, changes made to the database by DML or DDL operations. Database recovery applies redo entries to data files to reconstruct lost changes.

The database processes copy redo entries from the user memory space to the redo log buffer in the SGA. The redo entries take up continuous, sequential space in the buffer. The background process log writer process (LGWR) writes the redo log buffer to the active online redo log group on disk. Figure 16-8 shows this redo buffer activity.

**Figure 16-8 Redo Log Buffer**




LGWR writes redo sequentially to disk while DBW performs scattered writes of data blocks to disk. Scattered writes tend to be much slower than sequential writes. Because LGWR enable users to avoid waiting for DBW to complete its slow writes, the database delivers better performance.

The `LOG_BUFFER` initialization parameter specifies the amount of memory that Oracle Database uses when buffering redo entries. Unlike other SGA components, the redo log buffer and fixed SGA buffer do not divide memory into granules.

**See Also:**

- "Log Writer Process (LGWR)" and "Importance of Checkpoints for Instance Recovery"
- *Oracle Database Administrator's Guide* for information about the online redo log

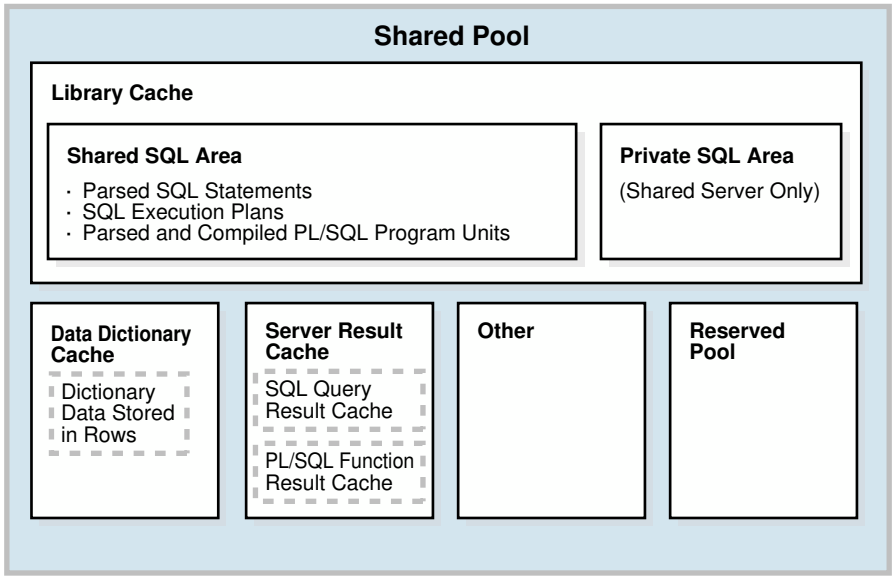
# Shared Pool

The **shared pool** caches various types of program data.

For example, the shared pool stores parsed SQL, PL/SQL code, system parameters, and data dictionary information. The shared pool is involved in almost every operation that occurs in the database. For example, if a user executes a SQL statement, then Oracle Database accesses the shared pool.

The shared pool is divided into several subcomponents, the most important of which are shown in Figure 16-9.

**Figure 16-9 Shared Pool**




This section includes the following topics:

- Library Cache
- Data Dictionary Cache
- Server Result Cache
- Reserved Pool

- Library Cache
  The **library cache** is a shared pool memory structure that stores executable SQL and PL/SQL code.
- Data Dictionary Cache
  The **data dictionary** is a collection of database tables and views containing reference information about the database, its structures, and its users.
- Server Result Cache
  The **server result cache** is a memory pool within the shared pool. Unlike the buffer pools, the server result cache holds result sets and not data blocks.
- Reserved Pool
  The **reserved pool** is a memory area in the shared pool that Oracle Database can use to allocate large contiguous chunks of memory.

## Library Cache

The **library cache** is a shared pool memory structure that stores executable SQL and PL/SQL code.

This cache contains the shared SQL and PL/SQL areas and control structures such as locks and library cache handles. In a shared server architecture, the library cache also contains private SQL areas.

When a SQL statement is executed, the database attempts to reuse previously executed code. If a parsed representation of a SQL statement exists in the library cache and can be shared, then the database reuses the code, known as a soft parse or a *library cache hit*. Otherwise, the database must build a new executable version of the application code, known as a hard parse or a *library cache miss*.

- Shared SQL Areas
  The database represents each SQL statement that it runs in the shared SQL area and private SQL area.
- Program Units and the Library Cache
  The library cache holds executable forms of PL/SQL programs and Java classes. These items are collectively referred to as *program units*.
- Allocation and Reuse of Memory in the Shared Pool
  The database allocates shared pool memory when a new SQL statement is parsed, unless the statement is DDL, which is not considered sharable. The size of memory allocated depends on the complexity of the statement.

### Shared SQL Areas

The database represents each SQL statement that it runs in the shared SQL area and private SQL area.

The database uses the shared SQL area to process the first occurrence of a SQL statement. This area is accessible to all users and contains the statement parse tree and execution plan. Only one shared SQL area exists for a unique statement. Each session issuing a SQL statement has a private SQL area in its PGA. Each user that submits the same statement has a private SQL area pointing to the same shared SQL area. Thus, many private SQL areas in separate PGAs can be associated with the same shared SQL area.

The database automatically determines when applications submit similar SQL statements. The database considers both SQL statements issued directly by users and applications and recursive SQL statements issued internally by other statements.

The database performs the following steps:

1. Checks the shared pool to see if a shared SQL area exists for a syntactically and semantically identical statement:
   - If an identical statement exists, then the database uses the shared SQL area for the execution of the subsequent new instances of the statement, thereby reducing memory consumption.
   - If an identical statement does not exist, then the database allocates a new shared SQL area in the shared pool. A statement with the same syntax but different semantics uses a child cursor.

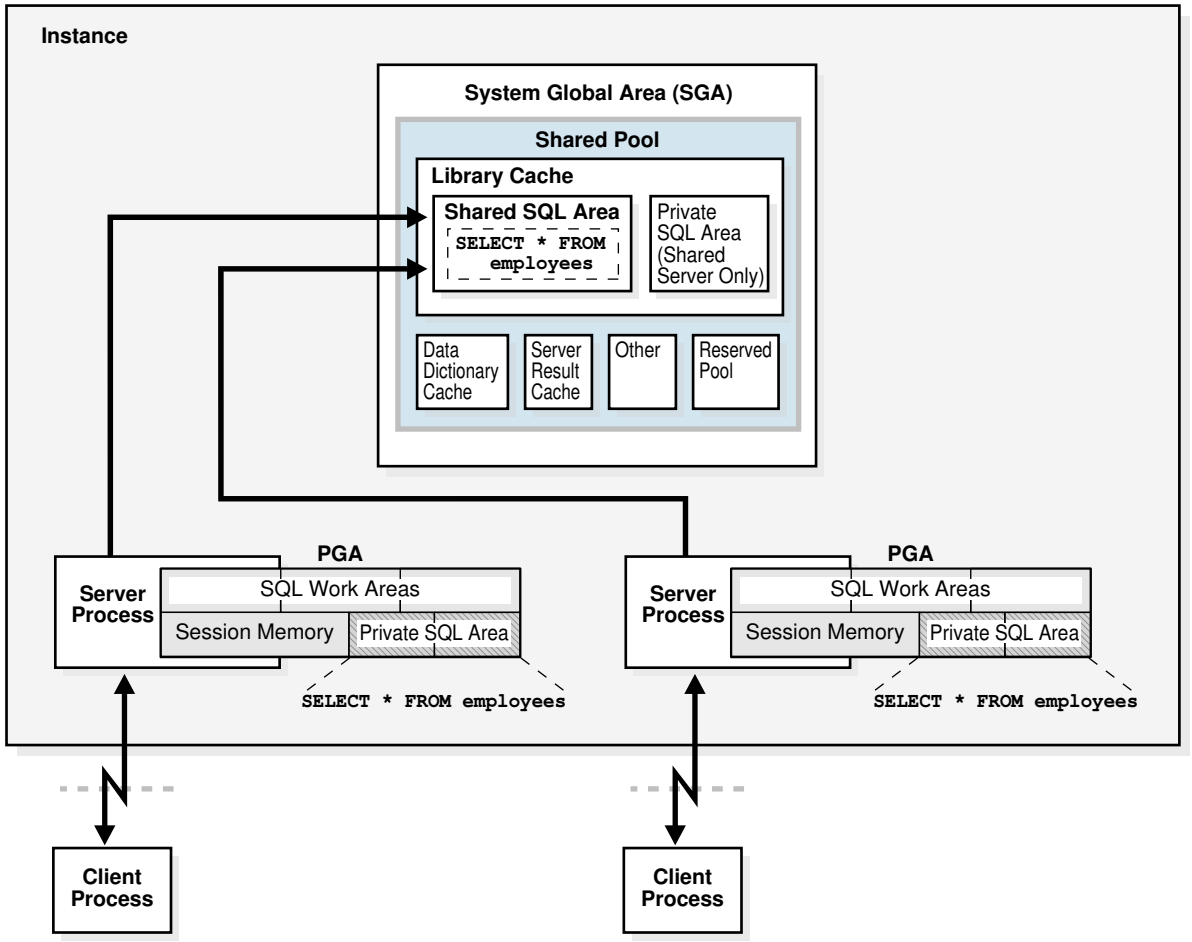
   In either case, the private SQL area for the user points to the shared SQL area that contains the statement and execution plan.

2. Allocates a private SQL area on behalf of the session

   The location of the private SQL area depends on the connection established for the session. If a session is connected through a shared server, then part of the private SQL area is kept in the SGA.

Figure 16-10 shows a dedicated server architecture in which two sessions keep a copy of the same SQL statement in their own PGAs. In a shared server, this copy is in the UGA, which is in the large pool or in the shared pool when no large pool exists.

**Figure 16-10 Private SQL Areas and Shared SQL Area**




**See Also:**

- "Private SQL Area"
- *Oracle Database Performance Tuning Guide* to learn more about managing the library cache
- *Oracle Database Development Guide* for more information about shared SQL

## Program Units and the Library Cache

The library cache holds executable forms of PL/SQL programs and Java classes. These items are collectively referred to as *program units*.

The database processes program units similarly to SQL statements. For example, the database allocates a shared area to hold the parsed, compiled form of a PL/SQL program. The database allocates a private area to hold values specific to the session that runs the program, including local, global, and package variables, and buffers for executing SQL. If multiple users run the same program, then each user maintains a separate copy of his or her private SQL area, which holds session-specific values, and accesses a single shared SQL area.

The database processes individual SQL statements within a PL/SQL program unit as previously described. Despite their origins within a PL/SQL program unit, these SQL statements use a shared area to hold their parsed representations and a private area for each session that runs the statement.

## Allocation and Reuse of Memory in the Shared Pool

The database allocates shared pool memory when a new SQL statement is parsed, unless the statement is DDL, which is not considered sharable. The size of memory allocated depends on the complexity of the statement.

In general, an item in the shared pool stays until the database removes it according to a least recently used (LRU) algorithm. The database allows shared pool items used by many sessions to remain in memory as long as they are useful, even if the database process that created the item terminates. This mechanism minimizes the overhead and processing of SQL statements. If space is needed for new items, then the database frees memory consumed by infrequently used items.

The `ALTER SYSTEM FLUSH SHARED_POOL` statement removes all information in the shared pool, as does changing the global database name.

**See Also:**

- *Oracle Database SQL Tuning Guide* for an overview of the life cycle of a shared SQL area
- *Oracle Database SQL Language Reference* for information about using `ALTER SYSTEM FLUSH SHARED_POOL`
- *Oracle Database Reference* for information about `V$SQL` and `V$SQLAREA` dynamic views

# Data Dictionary Cache

The **data dictionary** is a collection of database tables and views containing reference information about the database, its structures, and its users.

Oracle Database accesses the data dictionary frequently during SQL statement parsing. The data dictionary is accessed so often by Oracle Database that the following special memory locations are designated to hold dictionary data:

- Data dictionary cache

This cache holds information about database objects. The cache is also known as the *row cache* because it holds data as rows instead of buffers.

- Library cache

All server processes share these caches for access to data dictionary information.

> **See Also:**
>
> - "Data Dictionary and Dynamic Performance Views"
> - *Oracle Database Performance Tuning Guide* to learn how to allocate additional memory to the data dictionary cache

## Server Result Cache

The **server result cache** is a memory pool within the shared pool. Unlike the buffer pools, the server result cache holds result sets and not data blocks.

> **Note:**
>
> A client result cache differs from the server result cache. A client cache is configured at the application level and is located in client memory, not in database memory.

- SQL Query Result Cache
  The **SQL query result cache** is a subset of the server result cache that stores the results of queries and query fragments. You can enable or disable result caching at the database or statement level.
- PL/SQL Function Result Cache
  The **PL/SQL function result cache** is a subset of the server result cache that stores function result sets.
- Server Result Cache Infrastructure
  The query result cache and PL/SQL function result cache share the same infrastructure.
- Server Result Cache User Interface
  The `RESULT_CACHE_MODE` initialization parameter determines whether the SQL query result cache is used for all possible queries or only for hinted queries.

### SQL Query Result Cache

The **SQL query result cache** is a subset of the server result cache that stores the results of queries and query fragments. You can enable or disable result caching at the database or statement level.

When a query executes, the database determines whether the result exists in the query result cache. If the result is not cached, and if caching is enabled for the query, then the database runs the query, returns the result, and then caches it. If the result exists, however, then the database retrieves it from the cache instead of executing the query.

The database automatically invalidates a cached result whenever a transaction modifies the data or metadata of database objects used to construct the result. The next query cannot use

the cached result, so the database automatically computes a new result, and then caches it for use by subsequent queries. The cache refresh process is transparent to the application.

In effect, the query result cache functions as a "just-in-time" materialized view that the database creates and maintains as needed. The cache enables the database to avoid the expensive operation of rereading data blocks and recomputing results. Most applications benefit from this performance improvement.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn how to optimize response time using the query result cache

## PL/SQL Function Result Cache

The **PL/SQL function result cache** is a subset of the server result cache that stores function result sets.

Without caching, 1000 calls of a PL/SQL function at 1 second per call would take 1000 seconds. With caching, 1000 function calls with the same inputs could take 1 second *total*. Good candidates for result caching are frequently invoked functions that depend on relatively static data.

PL/SQL function code can include a request to cache its results. Upon invocation of this function, the system checks the cache. If the cache contains the result from a previous function call with the same parameter values, then the system returns the result to the invoker and does not reexecute the function body. If the cache does not contain the result, then the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the invoker.

> **Note:**
>
> You can specify the database objects used to compute a cached result, so that if any of them are updated, the cached result becomes invalid and must be recomputed.

A function that is invoked frequently with different arguments may generate results that are rarely reused, leading to performance degradation. Oracle Database tracks recently used PL/SQL functions that have the `RESULT_CACHE` hint. Using this history, the database only caches a PL/SQL function and argument pair if it has seen it *x* times in recent history, where *x* is set by an internal threshold. If the database needs more memory because too many results are cached, then one or more cached results are aged out.

`RESULT_CACHE_EXECUTION_THRESHOLD` specifies the number of times a function and a particular set of arguments must be seen until it is cached. Note that functions are considered unique if they have different arguments, for example, `MYFUNC(1,2)` and `MYFUNC(1,3)`. You can only set this parameter at the system level, not the session level.

> **See Also:**
>
> - *Oracle Database Development Guide* to learn more about the PL/SQL function result cache
> - *Oracle Database PL/SQL Language Reference* to learn more about the PL/SQL function result cache

## Server Result Cache Infrastructure

The query result cache and PL/SQL function result cache share the same infrastructure.

#### Object Types: Result, Dependency, and Temp

Within the server result cache, a **result object** is a data structure that stores the rows returned by a query or PL/SQL function. A **dependency object** stores the metadata for objects referred to by queries stored as result objects.

A typical query result is small, expensive to compute, reused often, and based on non-volatile tables. However, results can sometimes be large. If a query result passes the size limit, then the database stores part of the result as a **temp object** and part as a standard result object. A temp object is a collection of result cache metadata that points to a temporary tablespace segment that holds the actual query result.

#### Result Subcaches

Depending on the workload, the database may create result subcaches to support higher concurrency during read/write access. Oracle Database uses an internal algorithm to decide whether to create subcaches, and if so, how many to create.

## Server Result Cache User Interface

The `RESULT_CACHE_MODE` initialization parameter determines whether the SQL query result cache is used for all possible queries or only for hinted queries.

The `RESULT_CACHE_MAX_TEMP_SIZE` initialization parameter controls the maximum amount of temporary tablespace memory that the result cache can consume in the PDB. `RESULT_CACHE_MAX_TEMP_RESULT` controls the maximum amount of temporary tablespace memory that one cached query can consume.

To override the `RESULT_CACHE_MODE` parameter setting, you can annotate a query or query fragment with a `RESULT_CACHE` hint. Starting in Oracle Database 21c, the hint accepts the option `TEMP={TRUE|FALSE}`, which controls whether the results can be stored in a temporary tablespace.

The `DBMS_RESULT_CACHE` package enables you to administer that part of the shared pool that is used by the SQL result cache and the PL/SQL function result cache. You can invalidate objects, blacklist queries and objects, and perform related operations.

The `V$RESULT_CACHE_OBJECTS.TYPE` column indicates the type of objects in the cache. If query results are stored in temporary tablespace, then `TYPE` shows two objects: `Result` and `Temp`. Subcache metadata is accessible using `V$RESULT_SUBCACHE_STATISTICS` and related views.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about sizing and controlling the behavior of the result cache
> - *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_RESULT_CACHE` package
> - *Oracle Database Performance Tuning Guide* for more information about the client result cache
> - *Oracle Database Reference* to learn more about the `RESULT_CACHE_MODE` initialization parameter
> - *Oracle Database SQL Language Reference* to learn about the `RESULT_CACHE` hint

## Reserved Pool

The **reserved pool** is a memory area in the shared pool that Oracle Database can use to allocate large contiguous chunks of memory.

The database allocates memory from the shared pool in chunks. Chunking allows large objects (over 5 KB) to be loaded into the cache without requiring a single contiguous area. In this way, the database reduces the possibility of running out of contiguous memory because of fragmentation.

Infrequently, Java, PL/SQL, or SQL cursors may make allocations out of the shared pool that are larger than 5 KB. To allow these allocations to occur most efficiently, the database segregates a small amount of the shared pool for the reserved pool.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* to learn how to configure the reserved pool

# Large Pool

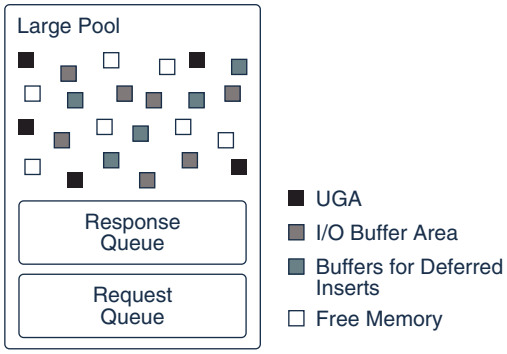
The **large pool** is an optional memory area intended for memory allocations that are larger than is appropriate for the shared pool.

The large pool can provide large memory allocations for the following:

- UGA for the shared server and the Oracle XA interface (used where transactions interact with multiple databases)
- Message buffers used in parallel execution
- Buffers for Recovery Manager (RMAN) I/O slaves
- Buffers for deferred inserts (inserts with the `MEMOPTIMIZE_WRITE` hint)

The following figure depicts the large pool.

**Figure 16-11 Large Pool**




- Large Pool Memory Management
  The large pool manages memory differently from the shared pool, which uses an LRU list so that portions of memory can age out.
- Large Pool Buffers for Deferred Inserts
  For a special type of insert known as a **deferred insert**, the database allocates buffers from the large pool.

> **See Also:**
>
> - "Query Coordinator" for information about allocating memory for parallel execution
> - "Dispatcher Request and Response Queues" to learn about allocating session memory for shared server
> - *Oracle Database Development Guide* to learn about Oracle XA
> - *Oracle Database Performance Tuning Guide* for more information about the large pool

## Large Pool Memory Management

The large pool manages memory differently from the shared pool, which uses an LRU list so that portions of memory can age out.

The large pool does not have an LRU list. When the database allocates large pool memory to a database session, this memory is not eligible to be released unless the session releases it. As soon as a portion of memory is freed, other processes can use it. By allocating session memory from the large pool, the database avoids the fragmentation that can occur in the shared pool.

## Large Pool Buffers for Deferred Inserts

For a special type of insert known as a **deferred insert**, the database allocates buffers from the large pool.

For rapid “fire and forget” inserts from Internet of Things (IoT) applications, the database infrastructure creates performance overhead. For example, the code path includes buffer

cache navigation, buffer pins, and concurrency protections. Array inserts minimize these costs, but the array must be built on the client side, which is not typical for Internet of Things applications. To address this issue, an Oracle application can use a hint to insert rows into a table specified as `MEMOPTIMIZE FOR WRITE`.

The inserts are deferred because they are buffered in the large pool, and then later written to disk asynchronously by background processes. The database processes deferred inserts as follows:

1. The application sends `MEMOPTIMIZE_WRITE` inserts to a middle tier, which can aggregate the data. While an IoT application will almost always send inserts to the middle tier, it is also possible to send inserts directly to the database. For example, using SQL*Plus sends the inserts directly to the database.
2. The middle tier writes the aggregation of inserts to the database server.
3. Optionally, the middle tier client retains a local copy of the data that it wrote in the previous step.
4. A server process writes data to a buffer or buffers in the large pool.

   To avoid contention, each buffer has its own internal locking mechanism. This locking mechanism is separate from the locking mechanism that the database buffer cache uses for its buffers. The basic write process is as follows:

   a. After instance startup, the first `MEMOPTIMIZE_WRITE` insert allocates the buffers from the large pool.
   b. The writer chooses a buffer from the list of available buffers.
   c. If the chosen buffer is not locked, and if this buffer has free space, then the client writes to the buffer, stamping each buffer write with a session-specific sequence number. If not, then the writer returns to the preceding step, and continues in this way until either a buffer is found or sufficient space has been freed in the large pool.
5. The database creates a server-side array from the buffered data.
6. The Space Management Coordinator (SMCO) and its helper processes (W*nnn*) write the array to disk asynchronously using the standard data block format.

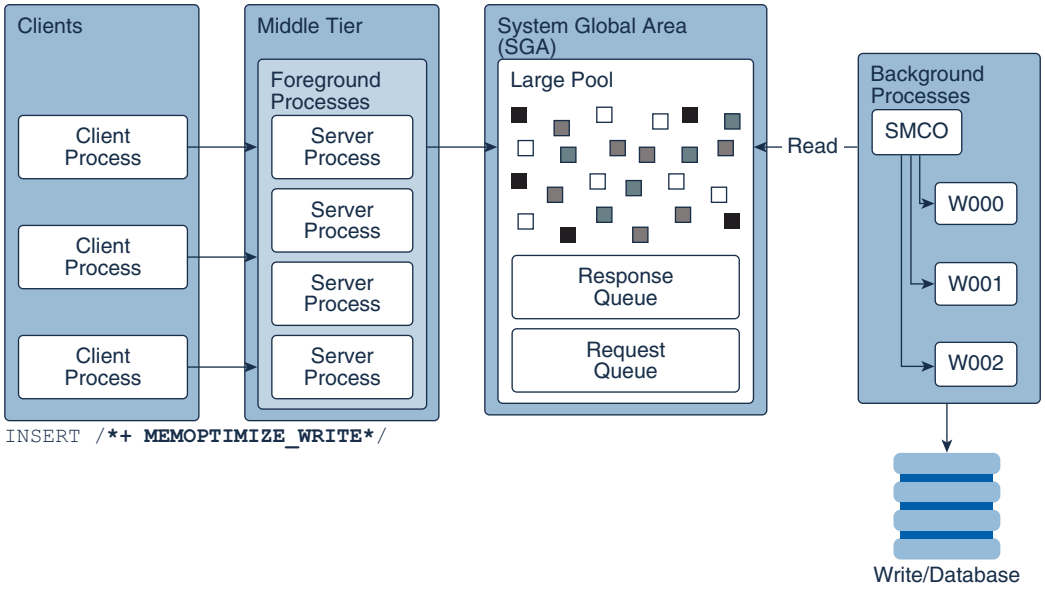
   Unlike standard inserts, deferred inserts are automatically committed and cannot be rolled back. The database commits the inserts to a given object in the order in which they appear within a session. There is no guarantee of ordering *between* objects or sessions.

   The database supports constraints and index maintenance just as for regular inserts. However, the database performs evaluations during the write to disk, not the write to the large pool.

> **Note:**
>
> For best performance, Oracle recommends disabling constraints.

The following figure depicts the workflow for deferred inserts.

**Figure 16-12 Deferred Insert Mechanism**




**See Also:**

- "Read Consistency and Deferred Inserts"
- *Oracle Database Performance Tuning Guide* to learn how to specify tables as `MEMOPTIMIZE FOR WRITE`
- *Oracle Database SQL Language Reference* for `INSERT` syntax and semantics
- *Oracle Database Reference* to learn more about `MEMOPTIMIZE_POOL_SIZE`

## Java Pool

The **Java pool** is an area of memory that stores all session-specific Java code and data within the Java Virtual Machine (JVM). This memory includes Java objects that are migrated to the Java session space at end-of-call.

For dedicated server connections, the Java pool includes the shared part of each Java class, including methods and read-only memory such as code vectors, but not the per-session Java state of each session. For shared server, the pool includes the shared part of each class and some UGA used for the state of each session. Each UGA grows and shrinks as necessary, but the total UGA size must fit in the Java pool space.

The Java Pool Advisor statistics provide information about library cache memory used for Java and predict how changes in the size of the Java pool can affect the parse rate. The Java Pool Advisor is internally turned on when `statistics_level` is set to `TYPICAL` or higher. These statistics reset when the advisor is turned off.

> **See Also:**
>
> - *Oracle Database Java Developer's Guide*
> - *Oracle Database Performance Tuning Guide* to learn about views containing Java pool advisory statistics

## Fixed SGA

The **fixed SGA** is an internal housekeeping area.

For example, the fixed SGA contains:

- General information about the state of the database and the instance, which the background processes need to access
- Information communicated between processes, such as information about locks

The size of the fixed SGA is set by Oracle Database and cannot be altered manually. The fixed SGA size can change from release to release.

> **See Also:**
>
> "Overview of Automatic Locks"

## Optional Performance-Related SGA Subareas

Some SGA subareas are only enabled for specific performance features.

This section contains the following topics:

- In-Memory Area
- Memoptimize Pool
- In-Memory Area
  The In-Memory Area is an optional SGA component that contains the **In-Memory Column Store** (IM column store).
- Memoptimize Pool
  The **memoptimize pool** stores buffers and related structures for heap-organized tables specified as `MEMOPTIMIZE FOR READ`.

### In-Memory Area

The In-Memory Area is an optional SGA component that contains the **In-Memory Column Store** (IM column store).

The IM column store contains copies of tables, partitions, and materialized views in a columnar format optimized for rapid scans. The IM column store supplements the database buffer cache, which stores data in traditional row format.

> **Note:**
>
> To enable an IM column store, you must have the Oracle Database In-Memory option.

> **See Also:**
>
> *Oracle Database In-Memory Guide* to learn more about the In-Memory Area and the IM column store

## Memoptimize Pool

The **memoptimize pool** stores buffers and related structures for heap-organized tables specified as `MEMOPTIMIZE FOR READ`.

This structure provides high performance and scalability for key-based queries such as `SELECT * FROM cust WHERE cid = 10`. To reduce end-to-end response time, clients pull requested buffers directly from the SGA over the network, avoiding CPU and operating system overhead. Applications can benefit from the memoptimize pool without requiring code changes.

The memoptimize pool contains two parts:

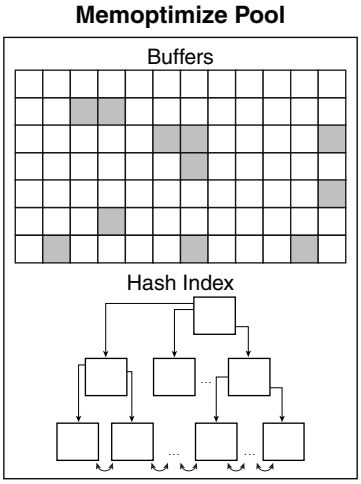
- Memoptimize buffer area

  To avoid disk I/O, the database permanently locks buffers for `MEMOPTIMIZE FOR READ` tables in the memoptimize pool, until the table is marked `NO MEMOPTIMIZE FOR READ`. The memoptimize buffers use the same structure as buffers in the database buffer cache. However, the buffers in the memoptimize pool are completely separate from the database buffer cache and do not count toward its size. The memoptimize buffer area occupies 75% of the memoptimize pool.

- Hash index

  A hash index is a non-persistent, segment data structure. The database allocates the hash index as multiple, noncontiguous memory units. Each unit contains a number of hash buckets. A separate map structure correlates a memory unit with a primary key. The hash index occupies 25% of the memoptimize pool.

**Figure 16-13 Memoptimize Pool**




To enable the memoptimize pool, set the `MEMOPTIMIZE_POOL_SIZE` initialization parameter to an integer value (the pool is disabled by default). The value specifies the amount of SGA to allocate to the pool. The `MEMOPTIMIZE_POOL_SIZE` value *does* count toward `SGA_TARGET`, but the database does *not* grow and shrink the memoptimize pool automatically. For example, if `SGA_TARGET` is 10 GB, and if `MEMOPTIMIZE_POOL_SIZE` is 1 GB, then a total of 9 GB is available for SGA memory other than the memoptimize pool.

To change the size of the memoptimize pool, you must set `MEMOPTIMIZE_POOL_SIZE` manually and restart the database instance. You cannot change the pool size dynamically using `ALTER SYSTEM`.

The `DBMS_MEMOPTIMIZE` package enables you to explicitly populate a table into the memoptimize pool.

**See Also:**

- *Oracle Database Performance Tuning Guide* to learn how to improve query performance by enabling the memoptimize pool
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_MEMOPTIMIZE` package
- *Oracle Database SQL Language Reference* to learn more about `CREATE TABLE ... MEMOPTIMIZE FOR READ`
- *Oracle Database Reference* to learn more about `MEMOPTIMIZE_POOL_SIZE`

# Overview of Software Code Areas

A **software code area** is a portion of memory that stores code that is being run or can be run. Oracle Database code is stored in a software area that is typically more exclusive and protected than the location of user programs.

Software areas are usually static in size, changing only when software is updated or reinstalled. The required size of these areas varies by operating system.

Software areas are read-only and can be installed shared or nonshared. Some database tools and utilities, such as Oracle Forms and SQL*Plus, can be installed shared, but some cannot. When possible, database code is shared so that all users can access it without having multiple copies in memory, resulting in reduced main memory and overall improvement in performance. Multiple instances of a database can use the same database code area with different databases if running on the same computer.

> **Note:**
>
> The option of installing software shared is not available for all operating systems, for example, on PCs operating Microsoft Windows. See your operating system-specific documentation for more information.

17

# Process Architecture

This chapter discusses the processes in an Oracle database.

This chapter contains the following sections:


- Introduction to Processes
- Overview of Client Processes
- Overview of Server Processes
- Overview of Background Processes


- Introduction to Processes
  A **process** is a mechanism in an operating system that can run a series of steps.
- Overview of Client Processes
  When a user runs an application such as a Pro*C program or SQL*Plus, the operating system creates a client process (sometimes called a *user process*) to run the user application. The client application has Oracle Database libraries linked into it that provide the APIs required to communicate with the database.
- Overview of Server Processes
  Oracle Database creates server processes to handle the requests of client processes connected to the instance. A client process always communicates with a database through a separate server process.
- Overview of Background Processes
  Background processes are additional processes used by a multiprocess Oracle database. The background processes perform maintenance tasks required to operate the database and to maximize performance for multiple users.

## Introduction to Processes

A **process** is a mechanism in an operating system that can run a series of steps.

The process execution architecture depends on the operating system. For example, on Windows an Oracle background process is a thread of execution within a process. On Linux and UNIX, an Oracle process is either an operating system process or a thread within an operating system process.

Processes run code modules. All connected Oracle Database users must run the following modules to access a database instance:

- Application or Oracle Database utility

  A database user runs a database application, such as a precompiler program or a database tool such as SQL*Plus, that issues SQL statements to a database.
- Oracle database code

  Each user has Oracle database code executing on his or her behalf that interprets and processes the application's SQL statements.

A process normally runs in its own private memory area. Most processes can periodically write to an associated trace file.

- Types of Processes
  A database instance contains or interacts with multiple processes.
- Multiprocess and Multithreaded Oracle Database Systems
  Multiprocess Oracle Database (also called multiuser Oracle Database) uses several processes to run different parts of the Oracle Database code and additional Oracle processes for the users—either one process for each connected user or one or more processes shared by multiple users.

> **See Also:**
>
> - "Trace Files"
> - *Oracle Database 2 Day Developer's Guide*

## Types of Processes

A database instance contains or interacts with multiple processes.

Processes are divided into the following types:

- A client process runs the application or Oracle tool code.
- An Oracle process is a unit of execution that runs the Oracle database code. In the multithreaded architecture, an Oracle process can be an operating system process or a thread within an operating system process. Oracle processes include the following subtypes:
  - A background process starts with the database instance and perform maintenance tasks such as performing instance recovery, cleaning up processes, writing redo buffers to disk, and so on.
  - A server process performs work based on a client request.

    For example, these processes parse SQL queries, place them in the shared pool, create and execute a query plan for each query, and read buffers from the database buffer cache or from disk.

    > **Note:**
    >
    > Server processes, and the process memory allocated in these processes, run in the database instance. The instance continues to function when server processes terminate.

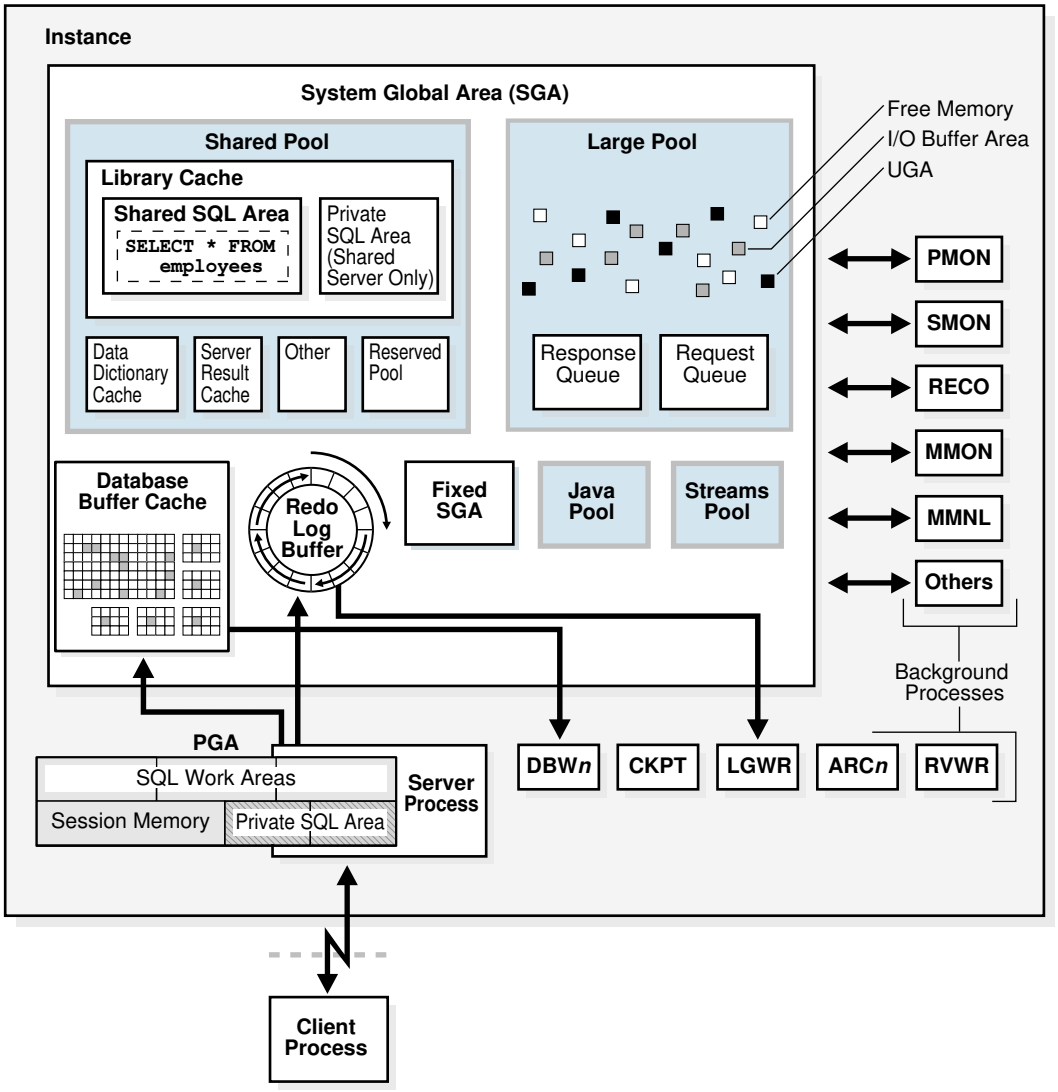
  - A slave process performs additional tasks for a background or server process.

The process structure varies depending on the operating system and the choice of Oracle Database options. For example, you can configure the code for connected users for dedicated server or shared server connections. In a shared server architecture, each server process that runs database code can serve multiple client processes.

Figure 17-1 shows a system global area (SGA) and background processes using dedicated server connections. For each user connection, a client process runs the application. This client process that is different from the dedicated server process that runs the database code. Each

client process is associated with its own server process, which has its own program global area (PGA).

**Figure 17-1 Oracle Processes and the SGA**




**See Also:**

- "Dedicated Server Architecture" and "Shared Server Architecture"
- Your Oracle Database operating system-specific documentation for more details on configuration choices
- *Oracle Database Reference* to learn about the `V$PROCESS` view

# Multiprocess and Multithreaded Oracle Database Systems

Multiprocess Oracle Database (also called multiuser Oracle Database) uses several processes to run different parts of the Oracle Database code and additional Oracle processes for the users—either one process for each connected user or one or more processes shared by multiple users.

Most databases are multiuser because a primary advantage of a database is managing data needed by multiple users simultaneously. Each process in a database instance performs a specific job. By dividing the work of the database and applications into several processes, multiple users and applications can connect to an instance simultaneously while the system gives good performance.

In releases earlier than Oracle Database 12c, Oracle processes did not run as threads on UNIX and Linux systems. Starting in Oracle Database 12c, the multithreaded Oracle Database model enables Oracle processes to execute as operating system threads in separate address spaces. When Oracle Database is installed, the database runs in process mode. You must set the `THREADED_EXECUTION` initialization parameter to `TRUE` to run the database in threaded mode. In threaded mode, some background processes on UNIX and Linux run as processes (with each process containing one thread), whereas the remaining Oracle processes run as threads within processes.

In a database running in threaded mode, PMON and DBW might run as operating system processes, whereas LGWR and CMON might run as threads within a single process. Two foreground processes and a parallel execution (PX) server process might run as threads in a second operating system process. A third operating system process might contain multiple foreground threads. Thus, "Oracle process" does not always mean "operating system process."

> **Note:**
>
> When the `THREADED_EXECUTION` initialization parameter is set to `TRUE`, operating system authentication is not supported.

**Example 17-1 Viewing Oracle Process Metadata**

The `V$PROCESS` view contains one row for each Oracle process connected to a database instance. For example, you can run the following query in SQL*Plus to get the operating system process ID and operating system thread ID for each process:

```
COL SPID FORMAT a8
COL STID FORMAT a8
SELECT SPID, STID, PROGRAM FROM V$PROCESS ORDER BY SPID;
```

The query yields the following partial sample output:

```
SPID   STID   PROGRAM
-----  -----  ----------------------------
7190   7190   oracle@samplehost (PMON)
7192   7192   oracle@samplehost (PSP0)
7194   7194   oracle@samplehost (VKTM)
7198   7198   oracle@samplehost (SCMN)
7198   7200   oracle@samplehost (GEN0)
```

```
7202   7202   oracle@samplehost (SCMN)
7202   7204   oracle@samplehost (DIAG)
7198   7205   oracle@samplehost (DBRM)
7202   7206   oracle@samplehost (DIA0)
.
.
.
```

**See Also:**

- *Oracle Database Performance Tuning Guide* to learn how to use the `V$PROCESS` view
- *Oracle Database Reference* to learn about the `THREADED_EXECUTION` initialization parameter

# Overview of Client Processes

When a user runs an application such as a Pro*C program or SQL*Plus, the operating system creates a client process (sometimes called a *user process*) to run the user application. The client application has Oracle Database libraries linked into it that provide the APIs required to communicate with the database.

- Client and Server Processes
  Client processes differ in important ways from the Oracle processes interacting directly with the instance.
- Connections and Sessions
  A database **connection** is a physical communication pathway between a client process and a database instance.
- Current Container
  A database session is a logical entity in the database instance memory that represents the state of a current user login to a database.
- Database Operations
  In the context of database monitoring, a **database operation** is session activity between two points in time, as defined by the end users or application code.

## Client and Server Processes

Client processes differ in important ways from the Oracle processes interacting directly with the instance.

The Oracle processes servicing the client process can read from and write to the SGA, whereas the client process cannot. A client process can run on a host other than the database host, whereas Oracle processes cannot.

For example, assume that a user on a client host starts SQL*Plus, and then connects over the network to database `sample` on a different host when the database instance is not started:

```
SQL> CONNECT SYS@inst1 AS SYSDBA
Enter password: *********
Connected to an idle instance.
```

On the client host, a search of the processes for either `sqlplus` or `sample` shows only the `sqlplus` client process:

```
% ps -ef | grep -e sample -e sqlplus | grep -v grep
clientuser 29437 29436  0 15:40 pts/1    00:00:00 sqlplus           as sysdba
```

On the database host, a search of the processes for either `sqlplus` or `sample` shows a server process with a nonlocal connection, but no client process:

```
% ps -ef | grep -e sample -e sqlplus | grep -v grep
serveruser 29441     1  0 15:40 ?        00:00:00 oraclesample (LOCAL=NO)
```

> **See Also:**
>
> "How an Instance Is Started" to learn how a client can connect to a database when the instance is not started

## Connections and Sessions

A database **connection** is a physical communication pathway between a client process and a database instance.

During a connection, a communication pathway is established using available interprocess communication mechanisms or network software. Typically, a connection occurs between a client process and a server process or dispatcher, but it can also occur between a client process and Oracle Connection Manager (CMAN).

## Current Container

A database session is a logical entity in the database instance memory that represents the state of a current user login to a database.

For example, when a user is authenticated by the database with a password, a session is established for this user. A session lasts from the time the user is authenticated by the database until the time the user disconnects or exits the database application.

### Sessions and Containers

For a given session, the **current container** in the CDB is the one in which a session is running. The current container can be the CDB root, an application root, or a PDB.

Each session has exactly one current container at any point in time. Because the data dictionary in each container is separate, Oracle Database uses the data dictionary in the current container for name resolution and privilege authorization.

### Relationship Between Connections and Sessions

A single connection can have 0, 1, or more sessions established on it. The sessions are independent: a commit in one session does not affect transactions in other sessions.

> **Note:**
>
> If Oracle Net connection pooling is configured, then it is possible for a connection to drop but leave the sessions intact.

Multiple sessions can exist concurrently for a single database user. As shown in the following figure, user `hr` can have multiple connections to a database. In dedicated server connections, the database creates a server process on behalf of each connection. Only the client process that causes the dedicated server to be created uses it. In a shared server connection, many client processes access a single shared server process.

**Figure 17-2 One Session for Each Connection**

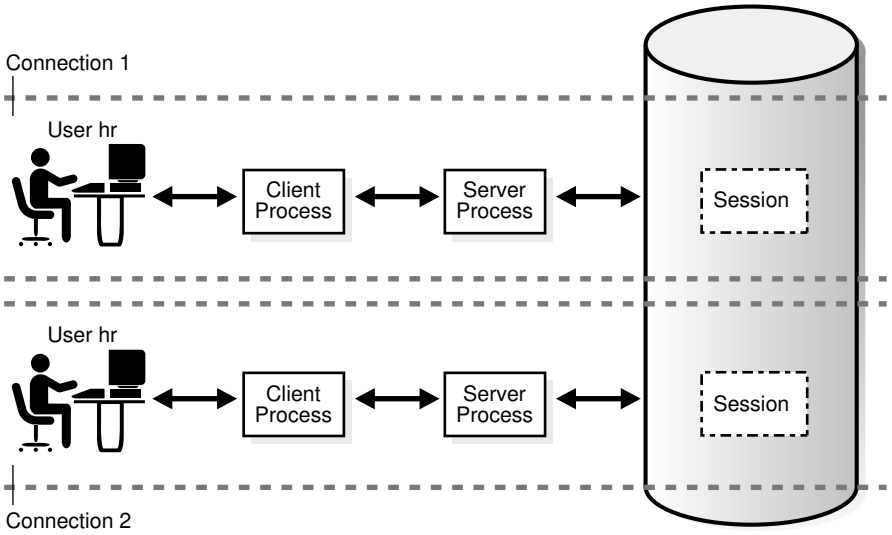
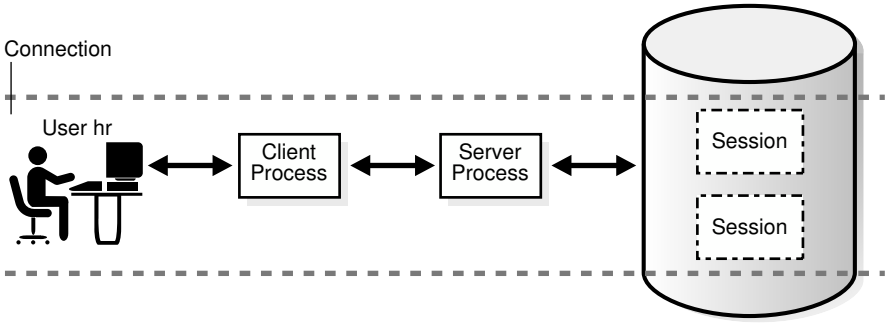



Figure 17-3 illustrates a case in which user `hr` has a single connection to a database, but this connection has two sessions.

**Figure 17-3 Two Sessions in One Connection**



Generating an autotrace report of SQL statement execution statistics re-creates the scenario in Figure 17-3.

The `DISCONNECT` command in Example 17-2 actually ends the *sessions*, not the connection.

**Example 17-2 Connections and Sessions**

The following example connects SQL*Plus to the database as user `SYSTEM` and enables tracing, thus creating a new session (sample output included):

```
SQL> SELECT SID, SERIAL#, PADDR FROM V$SESSION WHERE USERNAME = USER;

SID SERIAL# PADDR
--- ------- --------
 90      91 3BE2E41C

SQL> SET AUTOTRACE ON STATISTICS;
SQL> SELECT SID, SERIAL#, PADDR FROM V$SESSION WHERE USERNAME = USER;

SID SERIAL# PADDR
--- ------- --------
 88      93 3BE2E41C
 90      91 3BE2E41C
...
SQL> DISCONNECT
```

The `DISCONNECT` command actually ends the *sessions*, not the connection. Opening a new terminal and connecting to the instance as a different user, the following query shows that the connection with the address `3BE2E41C` is still active.

```
SQL> CONNECT dba1@inst1
Password: ********
Connected.
SQL> SELECT PROGRAM FROM V$PROCESS WHERE ADDR = HEXTORAW('3BE2E41C');

PROGRAM
------------------------------------------------
oracle@stbcs09-1 (TNS V1-V3)
```

> **See Also:**
>
> "Shared Server Architecture"

## Database Operations

In the context of database monitoring, a **database operation** is session activity between two points in time, as defined by the end users or application code.

A **simple database operation** is either a single SQL statement, or a single PL/SQL procedure or function. A **composite database operation** is a set of single or composite operations.

To monitor, compare, and tune tasks, you can divide a large set of tasks into database operations, and subdivide operations into phases. A use case is a PL/SQL batch job that is running slower than normal. By configuring the job as a database operation, you can identify and tune the expensive steps in the job.

Each execution of a database operation is uniquely identified by a pair of attributes: operation name and execution ID. One session can start or stop a database operation in a different session by specifying its session ID and serial number.

Two occurrences of the same database operation can execute at the same time using the same name but different execution IDs. Each execution of a database operation with the same name can contain different statements.

You create and manage database operations with the `DBMS_SQL_MONITOR` PL/SQL package. You can monitor operations using `V$SQL_MONITOR`, `V$SQL_PLAN_MONITOR`, and `V$SQL_MONITOR_SESSTAT`.

**See Also:**

- *Oracle Database SQL Tuning Guide* to learn how to monitor database operations
- *Oracle Database Data Warehousing Guide* to learn how to monitor long-running loads
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_SQL_MONITOR`
- *Oracle Database Reference* to learn more about `V$SQL_MONITOR`

# Overview of Server Processes

Oracle Database creates server processes to handle the requests of client processes connected to the instance. A client process always communicates with a database through a separate server process.

Server processes created on behalf of a database application can perform one or more of the following tasks:

- Parse and run SQL statements issued through the application, including creating and executing the query plan
- Execute PL/SQL code
- Read data blocks from data files into the database buffer cache (the DBW background process has the task of writing modified blocks back to disk)
- Return results in such a way that the application can process the information

- Dedicated Server Processes
  In dedicated server connections, the client connection is associated with one and only one server process.
- Shared Server Processes
  In shared server connections, client applications connect over a network to a **dispatcher process**, not a server process. For example, 20 client processes can connect to a single dispatcher process.

- How Oracle Database Creates Server Processes
  The database creates server processes in various ways, depending on the connection methods.

> **See Also:**
>
> "Stages of SQL Processing"

## Dedicated Server Processes

In dedicated server connections, the client connection is associated with one and only one server process.

On Linux, 20 client processes connected to a database instance are serviced by 20 server processes. Each client process communicates directly with its server process. This server process is dedicated to its client process for the duration of the session. The server process stores process-specific information and the UGA in its PGA.

> **See Also:**
>
> - "Dedicated Server Architecture"
> - "PGA Usage in Dedicated and Shared Server Modes"

## Shared Server Processes

In shared server connections, client applications connect over a network to a **dispatcher process**, not a server process. For example, 20 client processes can connect to a single dispatcher process.

The dispatcher process receives requests from connected clients and puts them into a request queue in the large pool. The first available shared server process takes the request from the queue and processes it. Afterward, the shared server places the result into the dispatcher response queue. The dispatcher process monitors this queue and transmits the result to the client.

Like a dedicated server process, a shared server process has its own PGA. However, the UGA for a session is in the SGA so that any shared server can access session data.

> **See Also:**
>
> - "Shared Server Architecture"
> - "Large Pool"

## How Oracle Database Creates Server Processes

The database creates server processes in various ways, depending on the connection methods.

The connection methods are as follows:

- Bequeath

  SQL*Plus, an OCI client, or another client application directly spawns the server process.
- Oracle Net listener

  The client application connects to the database through a listener.
- Dedicated broker

  This is a database process that creates foreground processes. Unlike the listener, the broker resides within the database instance. When using a dedicated broker, the client connects to the listener, which then hands off the connection to the dedicated broker.

When a connection does *not* use bequeath, the database creates the server process as follows:

1. The client application requests a new connection from the listener or broker.
2. The listener or broker initiates the creation of a new process or thread.
3. The operating system creates the new process or thread.
4. Oracle Database initializes various components and notifications.
5. The database hands over the connection and connection-specific code.

Optionally, if you use of the dedicated broker connection method, then you can pre-create a pool of server processes with the `DBMS_PROCESS` package. In this case, the Process Manager (PMAN) background process monitors the pool of pre-created processes, which wait to be associated with a client request. When a connection requires a server process, the database skips Steps 2-4 of process creation and performs only Step 5. This optimization improves performance.

> **Note:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_PROCESS` PL/SQL package
> - *Oracle Database Reference* to learn about the PMAN background process

# Overview of Background Processes

Background processes are additional processes used by a multiprocess Oracle database. The background processes perform maintenance tasks required to operate the database and to maximize performance for multiple users.

Each background process has a separate task, but works with the other processes. For example, the LGWR process writes data from the redo log buffer to the online redo log. When a filled redo log file is ready to be archived, LGWR signals another process to archive the redo log file.

Oracle Database creates background processes automatically when a database instance starts. An instance can have many background processes, not all of which always exist in every database configuration. The following query lists the background processes running on your database:

```
SELECT PNAME
FROM   V$PROCESS
WHERE  PNAME IS NOT NULL
ORDER BY PNAME;
```

This section includes the following topics:

- Mandatory Background Processes
- Optional Background Processes
- Slave Processes

- Mandatory Background Processes
  Mandatory background processes are present in all typical database configurations.
- Optional Background Processes
  An optional background process is any background process not defined as mandatory.
- Slave Processes
  Slave processes are background processes that perform work on behalf of other processes.

> **See Also:**
>
> *Oracle Database Reference* for descriptions of all the background processes

## Mandatory Background Processes

Mandatory background processes are present in all typical database configurations.

These processes run by default in a read/write database instance started with a minimally configured initialization parameter file. A read-only database instance disables some of these processes.

This section describes the following mandatory background processes:

- Process Monitor Process (PMON) Group
- Process Manager (PMAN)
- Listener Registration Process (LREG)
- System Monitor Process (SMON)
- Database Writer Process (DBW)
- Log Writer Process (LGWR)
- Checkpoint Process (CKPT)
- Manageability Monitor Processes (MMON and MMNL)
- Recoverer Process (RECO)

- Process Monitor Process (PMON) Group
  The **PMON group** includes PMON, Cleanup Main Process (CLMN), and Cleanup Helper Processes (CL*nn*). These processes are responsible for the monitoring and cleanup of other processes.
- Process Manager (PMAN)
  **Process Manager (PMAN)** oversees several background processes including shared servers, pooled servers, and job queue processes.
- Listener Registration Process (LREG)
  The **listener registration process (LREG)** registers information about the database instance and dispatcher processes with the Oracle Net Listener.
- System Monitor Process (SMON)
  The **system monitor process (SMON)** is in charge of a variety of system-level cleanup duties.
- Database Writer Process (DBW)
  The **database writer process (DBW)** writes the contents of database buffers to data files. DBW processes write modified buffers in the database buffer cache to disk.
- Log Writer Process (LGWR)
  The **log writer process (LGWR)** manages the online redo log buffer.
- Checkpoint Process (CKPT)
  The **checkpoint process (CKPT)** updates the control file and data file headers with checkpoint information and signals DBW to write blocks to disk. Checkpoint information includes the checkpoint position, SCN, and location in online redo log to begin recovery.
- Manageability Monitor Processes (MMON and MMNL)
  The **manageability monitor process (MMON)** performs many tasks related to the **Automatic Workload Repository (AWR)**.
- Recoverer Process (RECO)
  In a distributed database, the **recoverer process (RECO)** automatically resolves failures in distributed transactions.

> **See Also:**
>
> - "Read/Write and Read-Only Instances"
> - *Oracle Database Reference* for descriptions of other mandatory processes, including MMAN, DIAG, VKTM, DBRM, and PSP0
> - *Oracle Real Application Clusters Administration and Deployment Guide* and *Oracle Clusterware Administration and Deployment Guide* for more information about background processes specific to Oracle RAC and Oracle Clusterware

## Process Monitor Process (PMON) Group

The **PMON group** includes PMON, Cleanup Main Process (CLMN), and Cleanup Helper Processes (CL*nn*). These processes are responsible for the monitoring and cleanup of other processes.

The PMON group oversees cleanup of the buffer cache and the release of resources used by a client process. For example, the PMON group is responsible for resetting the status of the active transaction table, releasing locks that are no longer required, and removing the process ID of terminated processes from the list of active processes.

The database must ensure that resources held by terminated processes are released so they are usable by other processes. Otherwise, process may end up blocked or stuck in contention.

- Process Monitor Process (PMON)
  The **process monitor (PMON)** detects the termination of other background processes. If a server or dispatcher process terminates abnormally, then the PMON group is responsible for performing process recovery. Process termination can have multiple causes, including operating system kill commands or `ALTER SYSTEM KILL SESSION` statements.
- Cleanup Main Process (CLMN)
  PMON delegates cleanup work to the cleanup main process (CLMN). The task of detecting abnormal termination remains with PMON.
- Cleanup Helper Processes (CLnn)
  CLMN delegates cleanup work to the CL*nn* helper processes.
- Database Resource Quarantine
  If a process or session terminates, then the PMON group releases the held resources to the database. In some cases, the PMON group can automatically quarantine corrupted, unrecoverable resources so that the database instance is not immediately forced to terminate.

## Process Monitor Process (PMON)

The **process monitor (PMON)** detects the termination of other background processes. If a server or dispatcher process terminates abnormally, then the PMON group is responsible for performing process recovery. Process termination can have multiple causes, including operating system kill commands or `ALTER SYSTEM KILL SESSION` statements.

## Cleanup Main Process (CLMN)

PMON delegates cleanup work to the cleanup main process (CLMN). The task of detecting abnormal termination remains with PMON.

CLMN periodically performs cleanup of terminated processes, terminated sessions, transactions, network connections, idle sessions, detached transactions, and detached network connections that have exceeded their idle timeout.

## Cleanup Helper Processes (CLnn)

CLMN delegates cleanup work to the CL*nn* helper processes.

The CL*nn* processes assist in the cleanup of terminated processes and sessions. The number of helper processes is proportional to the amount of cleanup work to be done and the current efficiency of cleanup.

A cleanup process can become blocked, which prevents it from proceeding to clean up other processes. Also, if multiple processes require cleanup, then cleanup time can be significant. For these reasons, Oracle Database can use multiple helper processes in parallel to perform cleanup, thus alleviating slow performance.

The `V$CLEANUP_PROCESS` and `V$DEAD_CLEANUP` views contain metadata about CLMN cleanup. The `V$CLEANUP_PROCESS` view contains one row for every cleanup process. For example, if `V$CLEANUP_PROCESS.STATE` is `BUSY`, then the process is currently engaged in cleanup.

> **See Also:**
>
> *Oracle Database Reference* to learn more about `V$CLEANUP_PROCESS`

### Database Resource Quarantine

If a process or session terminates, then the PMON group releases the held resources to the database. In some cases, the PMON group can automatically quarantine corrupted, unrecoverable resources so that the database instance is not immediately forced to terminate.

The PMON group continues to perform as much cleanup as possible on the process or session that was holding the quarantined resource. The `V$QUARANTINE` view contains metadata such as the type of resource, amount of memory consumed, Oracle error causing the quarantine, and so on.

> **See Also:**
>
> *Oracle Database Reference* to learn more about `V$QUARANTINE`

## Process Manager (PMAN)

**Process Manager (PMAN)** oversees several background processes including shared servers, pooled servers, and job queue processes.

PMAN monitors, spawns, and stops the following types of processes:

- Dispatcher and shared server processes
- Connection broker and pooled server processes for database resident connection pools
- Job queue processes
- Restartable background processes

## Listener Registration Process (LREG)

The **listener registration process (LREG)** registers information about the database instance and dispatcher processes with the Oracle Net Listener.

When an instance starts, LREG polls the listener to determine whether it is running. If the listener is running, then LREG passes it relevant parameters. If it is not running, then LREG periodically attempts to contact it.

> **Note:**
>
> In releases before Oracle Database 12c, PMON performed the listener registration.

> **See Also:**
>
> "The Oracle Net Listener"

## System Monitor Process (SMON)

The **system monitor process (SMON)** is in charge of a variety of system-level cleanup duties.

Duties assigned to SMON include:

- Performing instance recovery, if necessary, at instance startup. In an Oracle RAC database, the SMON process of one database instance can perform instance recovery for a failed instance.
- Recovering terminated transactions that were skipped during instance recovery because of file-read or tablespace offline errors. SMON recovers the transactions when the tablespace or file is brought back online.
- Cleaning up unused temporary segments. For example, Oracle Database allocates extents when creating an index. If the operation fails, then SMON cleans up the temporary space.
- Coalescing contiguous free extents within dictionary-managed tablespaces.

SMON checks regularly to see whether it is needed. Other processes can call SMON if they detect a need for it.

## Database Writer Process (DBW)

The **database writer process (DBW)** writes the contents of database buffers to data files. DBW processes write modified buffers in the database buffer cache to disk.

Although one database writer process (DBW0) is adequate for most systems, you can configure additional processes—DBW1 through DBW9, DBWa through DBWz, and BW36 through BW99—to improve write performance if your system modifies data heavily. These additional DBW processes are not useful on uniprocessor systems.

The DBW process writes dirty buffers to disk under the following conditions:

- When a server process cannot find a clean reusable buffer after scanning a threshold number of buffers, it signals DBW to write. DBW writes dirty buffers to disk asynchronously if possible while performing other processing.
- DBW periodically writes buffers to advance the checkpoint, which is the position in the redo thread from which instance recovery begins. The log position of the checkpoint is determined by the oldest dirty buffer in the buffer cache.

In many cases the blocks that DBW writes are scattered throughout the disk. Thus, the writes tend to be slower than the sequential writes performed by LGWR. DBW performs multiblock writes when possible to improve efficiency. The number of blocks written in a multiblock write varies by operating system.

> **See Also:**
>
> - "Database Buffer Cache"
> - "Overview of Checkpoints"
> - *Oracle Database Performance Tuning Guide* for advice on configuring, monitoring, and tuning DBW

## Log Writer Process (LGWR)

The **log writer process (LGWR)** manages the online redo log buffer.

LGWR writes one portion of the buffer to the online redo log. By separating the tasks of modifying database buffers, performing scattered writes of dirty buffers to disk, and performing fast sequential writes of redo to disk, the database improves performance.

In the following circumstances, LGWR writes all redo entries that have been copied into the buffer since the last time it wrote:

- A user commits a transaction.
- An online redo log switch occurs.
- Three seconds have passed since LGWR last wrote.
- The redo log buffer is one-third full or contains 1 MB of buffered data.
- DBW must write modified buffers to disk.

  Before DBW can write a dirty buffer, the database must write to disk the redo records associated with changes to the buffer (the write-ahead protocol). If DBW discovers that some redo records have not been written, it signals LGWR to write the records to disk, and waits for LGWR to complete before writing the data buffers to disk.

- LGWR and Commits
  Oracle Database uses a fast commit mechanism to improve performance for committed transactions.
- LGWR and Inaccessible Files
  LGWR writes synchronously to the active mirrored group of online redo log files.

> **See Also:**
>
> "Commits of Transactions"

### LGWR and Commits

Oracle Database uses a fast commit mechanism to improve performance for committed transactions.

When a user issues a `COMMIT` statement, the transaction is assigned a **system change number (SCN)**. LGWR puts a commit record in the redo log buffer and writes it to disk immediately, along with the commit SCN and transaction's redo entries.

The redo log buffer is circular. When LGWR writes redo entries from the redo log buffer to an online redo log file, server processes can copy new entries over the entries in the redo log buffer that have been written to disk. LGWR normally writes fast enough to ensure that space is always available in the buffer for new entries, even when access to the online redo log is heavy.

The atomic write of the redo entry containing the transaction's commit record is the single event that determines that the transaction has committed. Oracle Database returns a success code to the committing transaction although the data buffers have not yet been written to disk. The corresponding changes to data blocks are deferred until it is efficient for DBW to write them to the data files.

> **Note:**
>
> LGWR can write redo log entries to disk before a transaction commits. The changes that are protected by the redo entries become permanent only if the transaction later commits.

When activity is high, LGWR can use group commits. For example, a user commits, causing LGWR to write the transaction's redo entries to disk. During this write other users commit. LGWR cannot write to disk to commit these transactions until its previous write completes. Upon completion, LGWR can write the list of redo entries of waiting transactions (not yet committed) in one operation. In this way, the database minimizes disk I/O and maximizes performance. If commits requests continue at a high rate, then every write by LGWR can contain multiple commit records.

### LGWR and Inaccessible Files

LGWR writes synchronously to the active mirrored group of online redo log files.

If a log file is inaccessible, then LGWR continues writing to other files in the group and writes an error to the LGWR trace file and the alert log. If all files in a group are damaged, or if the group is unavailable because it has not been archived, then LGWR cannot continue to function.

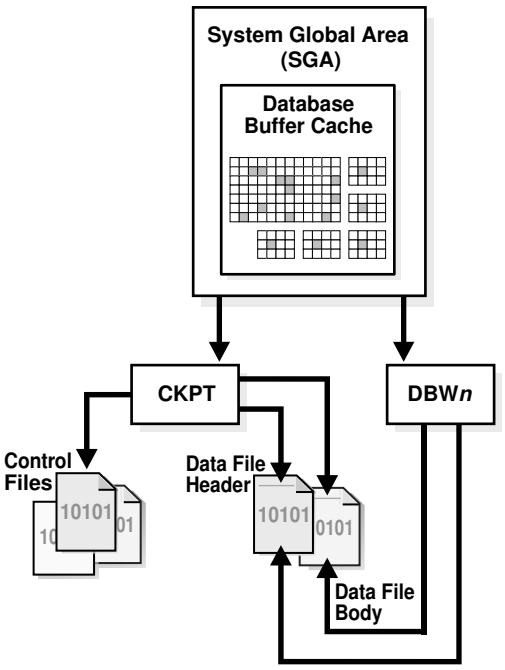
> **See Also:**
>
> - "How Oracle Database Writes to the Online Redo Log" and "Redo Log Buffer "
> - *Oracle Database Performance Tuning Guide*for information about how to monitor and tune the performance of LGWR

## Checkpoint Process (CKPT)

The **checkpoint process (CKPT)** updates the control file and data file headers with checkpoint information and signals DBW to write blocks to disk. Checkpoint information includes the checkpoint position, SCN, and location in online redo log to begin recovery.

As shown in Figure 17-4, CKPT does not write data blocks to data files or redo blocks to online redo log files.

**Figure 17-4 Checkpoint Process**




> **See Also:**
>
> "Overview of Checkpoints"

## Manageability Monitor Processes (MMON and MMNL)

The **manageability monitor process (MMON)** performs many tasks related to the **Automatic Workload Repository (AWR)**.

For example, MMON writes when a **metric** violates its threshold value, taking snapshots, and capturing statistics value for recently modified SQL objects.

The manageability monitor lite process (MMNL) writes statistics from the Active Session History (ASH) buffer in the SGA to disk. MMNL writes to disk when the ASH buffer is full.

> **See Also:**
>
> "*Oracle Database 2 Day + Performance Tuning Guide*" and
>
> "Overview of Active Session History"

## Recoverer Process (RECO)

In a distributed database, the **recoverer process (RECO)** automatically resolves failures in distributed transactions.

The RECO process of a node automatically connects to other databases involved in an in-doubt distributed transaction. When RECO reestablishes a connection between the databases, it automatically resolves all in-doubt transactions, removing from each database's pending transaction table any rows that correspond to the resolved transactions.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about transaction recovery in distributed systems

## Optional Background Processes

An optional background process is any background process not defined as mandatory.

Most optional background processes are specific to tasks or features. For example, background processes that support Oracle ASM are only available when this feature is enabled.

This section describes some common optional processes:

- Archiver Processes (ARC*n*)
- Job Queue Processes (CJQ0 and Jnnn)
- Flashback Data Archive Process (FBDA)
- Space Management Coordinator Process (SMCO)

- Archiver Processes (ARCn)
  An **archiver process (ARCn)** copies online redo log files to offline storage after a redo log switch occurs.
- Job Queue Processes (CJQ0 and Jnnn)
  A **queue process** runs user jobs, often in batch mode. A job is a user-defined task scheduled to run one or more times.
- Flashback Data Archive Process (FBDA)
  The **flashback data archive process (FBDA)** archives historical rows of tracked tables into Flashback Data Archives.
- Space Management Coordinator Process (SMCO)
  The SMCO process coordinates the execution of various space management related tasks.

> **See Also:**
>
> - *Oracle Database Advanced Queuing User's Guide* for an introduction to Oracle Database Advanced Queuing (AQ)
> - *Oracle Database Reference* for descriptions of background processes specific to AQ and Oracle ASM

## Archiver Processes (ARC*n*)

An **archiver process (ARCn)** copies online redo log files to offline storage after a redo log switch occurs.

These processes can also collect transaction redo data and transmit it to standby database destinations. ARC*n* processes exist *only* when the database is in ARCHIVELOG mode and automatic archiving is enabled.

> **See Also:**
>
> - "Archived Redo Log Files"
> - *Oracle Database Administrator's Guide* to learn how to adjust the number of archiver processes
> - *Oracle Database Performance Tuning Guide* to learn how to tune archiver performance

## Job Queue Processes (CJQ0 and Jnnn)

A **queue process** runs user jobs, often in batch mode. A job is a user-defined task scheduled to run one or more times.

For example, you can use a job queue to schedule a long-running update in the background. Given a start date and a time interval, the job queue processes attempt to run the job at the next occurrence of the interval.

Oracle Database manages job queue processes dynamically, thereby enabling job queue clients to use more job queue processes when required. The database releases resources used by the new processes when they are idle.

Dynamic job queue processes can run many jobs concurrently at a given interval. The sequence of events is as follows:

1. The job coordinator process (CJQ0) is automatically started and stopped as needed by Oracle Scheduler. The coordinator process periodically selects jobs that need to be run from the system `JOB$` table. New jobs selected are ordered by time.
2. The coordinator process dynamically spawns job queue slave processes (Jnnn) to run the jobs.
3. The job queue process runs one of the jobs that was selected by the CJQ0 process for execution. Each job queue process runs one job at a time to completion.
4. After the process finishes execution of a single job, it polls for more jobs. If no jobs are scheduled for execution, then it enters a sleep state, from which it wakes up at periodic intervals and polls for more jobs. If the process does not find any new jobs, then it terminates after a preset interval.

The initialization parameter `JOB_QUEUE_PROCESSES` represents the maximum number of job queue processes that can concurrently run on an instance. However, clients should not assume that all job queue processes are available for job execution.

> **Note:**
>
> The coordinator process is not started if the initialization parameter `JOB_QUEUE_PROCESSES` is set to 0.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* to learn about Oracle Scheduler
> - *Oracle Database Advanced Queuing User's Guide* to learn about AQ background processes

## Flashback Data Archive Process (FBDA)

The **flashback data archive process (FBDA)** archives historical rows of tracked tables into Flashback Data Archives.

When a transaction containing DML on a tracked table commits, this process stores the pre-image of the changed rows into the Flashback Data Archive. It also keeps metadata on the current rows.

FBDA automatically manages the Flashback Data Archive for space, organization, and retention. Additionally, the process keeps track of how long the archiving of tracked transactions has occurred.

## Space Management Coordinator Process (SMCO)

The SMCO process coordinates the execution of various space management related tasks.

Typical tasks include proactive space allocation and space reclamation. SMCO dynamically spawns slave processes (W*nnn*) to implement the task.

> **See Also:**
>
> *Oracle Database Development Guide* to learn about the Flashback Data Archive and the Temporal History feature

# Slave Processes

Slave processes are background processes that perform work on behalf of other processes.

This section describes some slave processes used by Oracle Database.

- I/O Slave Processes
  I/O slave processes (Innn) simulate asynchronous I/O for systems and devices that do not support it.

- Parallel Execution (PX) Server Processes
  In **parallel execution**, multiple processes work together simultaneously to run a single SQL statement.

> **See Also:**
>
> *Oracle Database Reference* for descriptions of Oracle Database slave processes

## I/O Slave Processes

I/O slave processes (Innn) simulate asynchronous I/O for systems and devices that do not support it.

In asynchronous I/O, there is no timing requirement for transmission, enabling other processes to start before the transmission has finished.

For example, assume that an application writes 1000 blocks to a disk on an operating system that does not support asynchronous I/O. Each write occurs sequentially and waits for a confirmation that the write was successful. With asynchronous disk, the application can write the blocks in bulk and perform other work while waiting for a response from the operating system that all blocks were written.

To simulate asynchronous I/O, one process oversees several slave processes. The invoker process assigns work to each of the slave processes, who wait for each write to complete and report back to the invoker when done. In true asynchronous I/O the operating system waits for the I/O to complete and reports back to the process, while in simulated asynchronous I/O the slaves wait and report back to the invoker.

The database supports different types of I/O slaves, including the following:

- I/O slaves for Recovery Manager (RMAN)

  When using RMAN to back up or restore data, you can use I/O slaves for both disk and tape devices.
- Database writer slaves

  If it is not practical to use multiple database writer processes, such as when the computer has one CPU, then the database can distribute I/O over multiple slave processes. DBW is the only process that scans the buffer cache LRU list for blocks to be written to disk. However, I/O slaves perform the I/O for these blocks.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* to learn more about I/O slaves for backup and restore operations
> - *Oracle Database Performance Tuning Guide* to learn more about database writer slaves

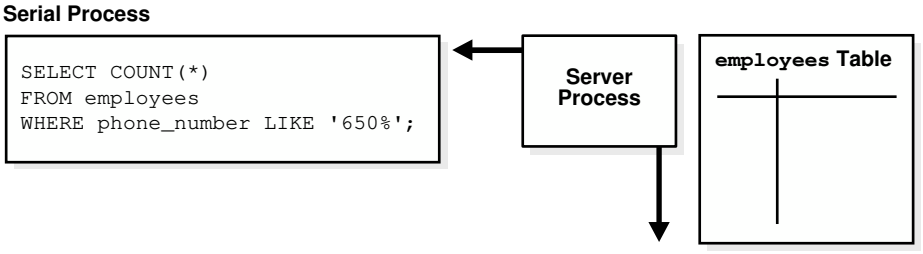
# Parallel Execution (PX) Server Processes

In **parallel execution**, multiple processes work together simultaneously to run a single SQL statement.

By dividing the work among multiple processes, Oracle Database can run the statement more quickly. For example, four processes handle four different quarters in a year instead of one process handling all four quarters by itself.

Parallel execution contrasts with serial execution, in which a single server process performs all necessary processing for the sequential execution of a SQL statement. For example, to perform a full table scan such as `SELECT * FROM employees`, one server process performs all of the work, as shown in Figure 17-5.

**Figure 17-5 Serial Full Table Scan**




Parallel execution reduces response time for data-intensive operations on large databases such as data warehouses. Symmetric multiprocessing (SMP) and clustered system gain the largest performance benefits from parallel execution because statement processing can be split up among multiple CPUs. Parallel execution can also benefit certain types of OLTP and hybrid systems.

In Oracle RAC systems, the service placement of a specific service controls parallel execution. Specifically, parallel processes run on the nodes on which the service is configured. By default, Oracle Database runs parallel processes only on an instance that offers the service used to connect to the database. This does not affect other parallel operations such as parallel recovery or the processing of `GV$` queries.

- Query Coordinator
  In parallel execution, the server process acts as the **query coordinator** (also called the *parallel execution coordinator*).
- Producers and Consumers
  Parallel execution servers are divided into producers and consumers. The producers are responsible for processing their data and then distributing it to the consumers that need it.
- Granules
  In parallel execution, a table is divided dynamically into load units. Each unit, called a **granule**, is the smallest unit of work when accessing data.

> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide* to learn more about parallel execution
> - *Oracle Real Application Clusters Administration and Deployment Guide* for considerations regarding parallel execution in Oracle RAC environments

## Query Coordinator

In parallel execution, the server process acts as the **query coordinator** (also called the *parallel execution coordinator*).

The query coordinator is responsible for the following:

1. Parsing the query
2. Allocating and controlling the parallel execution server processes
3. Sending output to the user

Given a query plan for a query, the coordinator breaks down each operator in a SQL query into parallel pieces, runs them in the order specified in the query, and integrates the partial results produced by the parallel execution servers executing the operators.

The number of parallel execution servers assigned to a single operation is the degree of parallelism for an operation. Multiple operations within the same SQL statement all have the same degree of parallelism.

## Producers and Consumers

Parallel execution servers are divided into producers and consumers. The producers are responsible for processing their data and then distributing it to the consumers that need it.

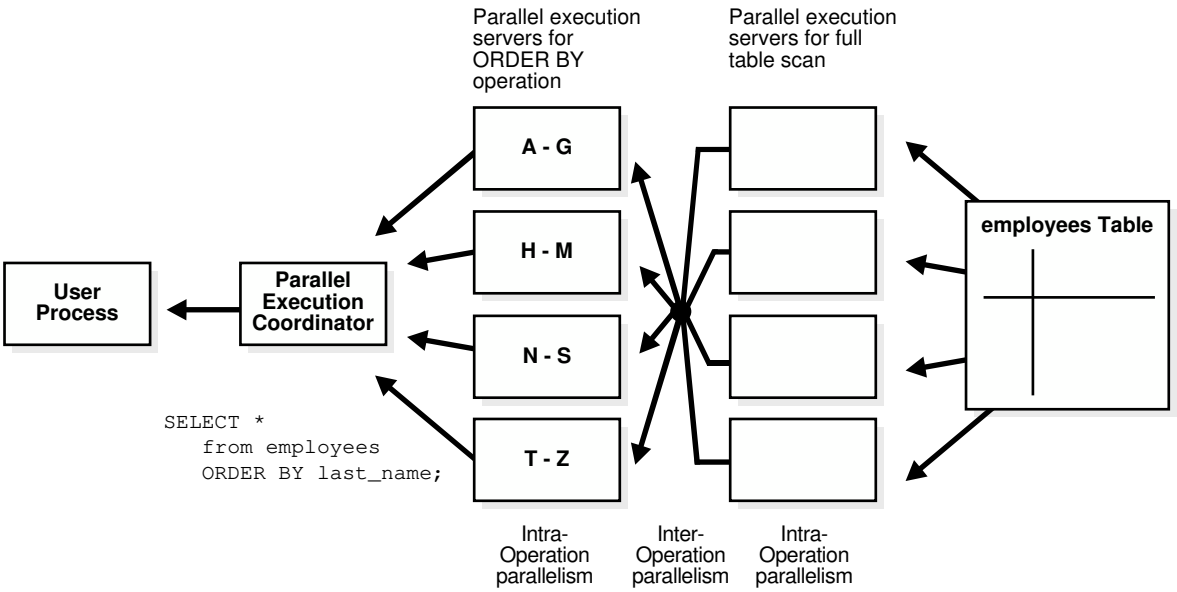
The database can perform the distribution using a variety of techniques. Two common techniques are a broadcast and a hash. In a broadcast, each producer sends the rows to all consumers. In a hash, the database computes a hash function on a set of keys and makes each consumer responsible for a subset of hash values.

Figure 17-6 represents the interplay between producers and consumers in the parallel execution of the following statement:

```
SELECT * FROM employees ORDER BY last_name;
```

The execution plan implements a full scan of the `employees` table. The scan is followed by a sort of the retrieved rows. All of the producer processes involved in the scan operation send rows to the appropriate consumer process performing the sort.

**Figure 17-6 Producers and Consumers**



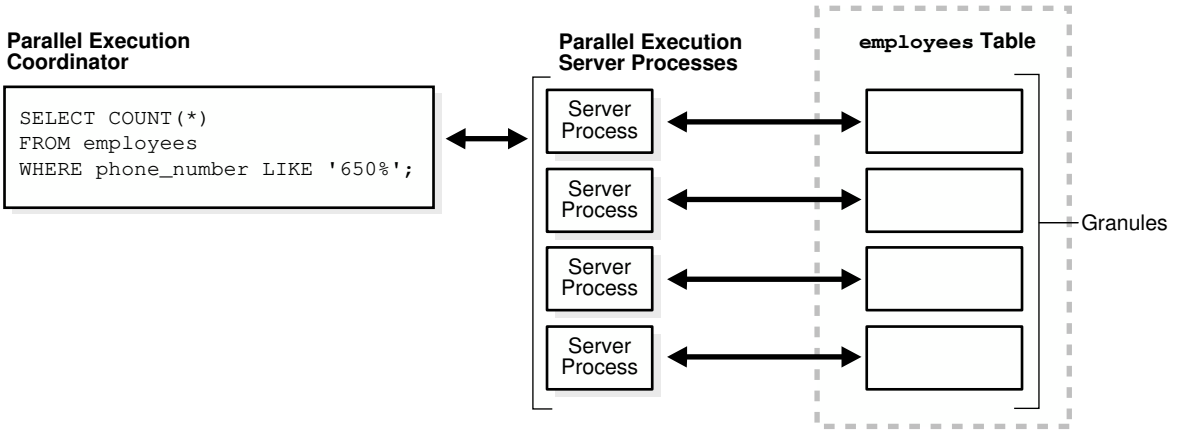

## Granules

In parallel execution, a table is divided dynamically into load units. Each unit, called a **granule**, is the smallest unit of work when accessing data.

A block-based granule is a range of data blocks of the table read by a single parallel execution server (also called a *PX server*), which uses `Pnnn` as a name format. To obtain an even distribution of work among parallel server processes, the number of granules is always much higher than the requested DOP.

Figure 17-7 shows a parallel scan of the `employees` table.

**Figure 17-7 Parallel Full Table Scan**




The database maps granules to parallel execution servers at execution time. When a parallel execution server finishes reading the rows corresponding to a granule, and when granules remain, it obtains another granule from the query coordinator. This operation continues until

the table has been read. The execution servers send results back to the coordinator, which assembles the pieces into the desired full table scan.

**See Also:**

- *Oracle Database VLDB and Partitioning Guide* to learn how to use parallel execution
- *Oracle Database Data Warehousing Guide* to learn about recommended initialization parameters for parallelism

18

# Application and Oracle Net Services Architecture

This chapter defines application architecture and describes how an Oracle database and database applications work in a distributed processing environment. This material applies to almost every type of Oracle Database environment.

This chapter contains the following sections:


- Overview of Oracle Application Architecture
- Overview of Oracle Net Services Architecture
- Overview of the Program Interface


- Overview of Oracle Application Architecture
  In the context of this chapter, **application architecture** refers to the computing environment in which a database application connects to an Oracle database.
- Overview of Oracle Net Services Architecture
  **Oracle Net Services** is a suite of networking components that provides enterprise-wide connectivity solutions in distributed, heterogeneous computing environments.
- Overview of the Program Interface
  The **program interface** is the software layer between a database application and Oracle Database.

## Overview of Oracle Application Architecture

In the context of this chapter, **application architecture** refers to the computing environment in which a database application connects to an Oracle database.

This section contains the following topics:


- Overview of Client/Server Architecture
- Overview of Multitier Architecture
- Overview of Grid Architecture


- Overview of Client/Server Architecture
  In the Oracle Database environment, the database application and the database are separated into a **client/server architecture**.
- Overview of Multitier Architecture
  In a traditional multitier architecture, an application server provides data for clients and serves as an interface between clients and database servers.
- Overview of Grid Architecture
  In an Oracle Database environment, **grid computing** is a computing architecture that effectively pools large numbers of servers and storage into a flexible, on-demand computing resource.

# Overview of Client/Server Architecture

In the Oracle Database environment, the database application and the database are separated into a **client/server architecture**.

The components are as follows:

- The client runs the database application, for example, SQL*Plus or a Visual Basic data entry program, that accesses database information and interacts with a user.
- The server runs the Oracle Database software and handles the functions required for concurrent, shared data access to an Oracle database.

Although the client application and database can run on the same computer, greater efficiency is often achieved when the client portions and server portion are run by different computers connected through a network. The following sections discuss variations in the Oracle Database client/server architecture.

- Distributed Processing
  Using multiple hosts to process an individual task is known as **distributed processing**.
- Advantages of a Client/Server Architecture
  Oracle Database client/server architecture in a distributed processing environment provides a number of benefits.

## Distributed Processing

Using multiple hosts to process an individual task is known as **distributed processing**.

Front-end and back-end processing occurs on different computers. In Figure 18-1, the client and server are located on different hosts connected through Oracle Net Services.

**Figure 18-1 Client/Server Architecture and Distributed Processing**

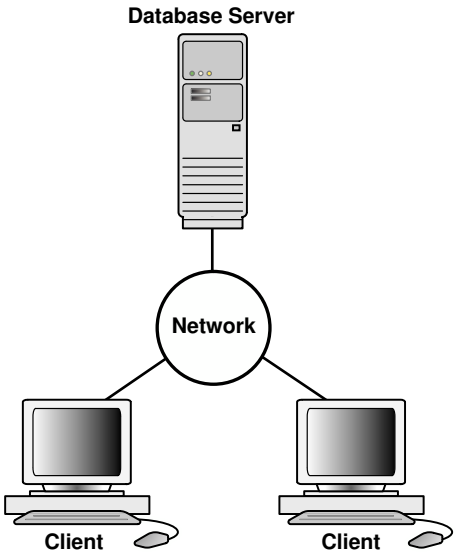
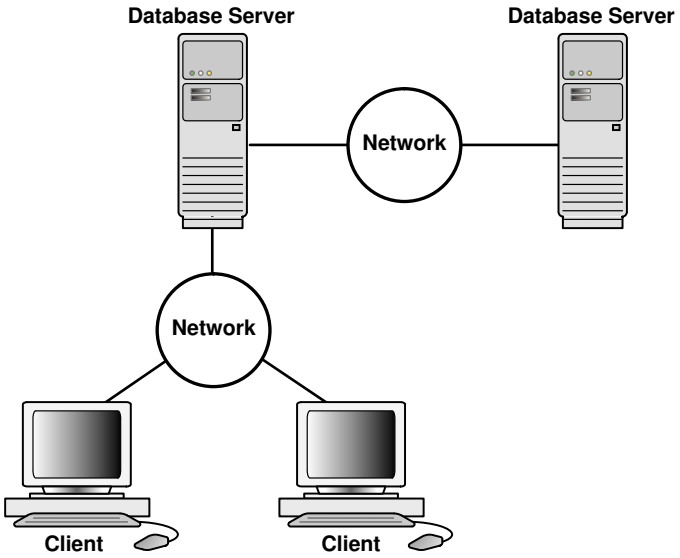



Figure 18-2 is a variation that depicts a distributed database. In this example, a database on one host accesses data on a separate database located on a different host.

**Figure 18-2 Client/Server Architecture and Distributed Database**




> **Note:**
>
> This rest of this chapter applies to environments with one database on one server.

## Advantages of a Client/Server Architecture

Oracle Database client/server architecture in a distributed processing environment provides a number of benefits.

Benefits include:

- Client applications are not responsible for performing data processing. Rather, they request input from users, request data from the server, and then analyze and present this data using the display capabilities of the client workstation or the terminal (for example, using graphics or spreadsheets).
- Client applications are not dependent on the physical location of the data. Even if the data is moved or distributed to other database servers, the application continues to function with little or no modification.
- Oracle Database exploits the multitasking and shared-memory facilities of its underlying operating system. Consequently, it delivers the highest possible degree of concurrency, data integrity, and performance to its client applications.
- Client workstations or terminals can be optimized for the presentation of data (for example, by providing graphics and mouse support), while the server can be optimized for the processing and storage of data (for example, by having large amounts of memory and disk space).
- In networked environments, you can use inexpensive client workstations to access the remote data of the server effectively.
- The database can scale as your system grows. You can add multiple servers to distribute the database processing load throughout the network (horizontally scaled), or you can move the database to a minicomputer or mainframe to take advantage of a larger system's

performance (vertically scaled). In either case, data and applications are maintained with little or no modification because Oracle Database is portable between systems.

- In networked environments, shared data is stored on the servers rather than on all computers, making it easier and more efficient to manage concurrent access.
- In networked environments, client applications submit database requests to the server using SQL statements. After it is received, each SQL statement is processed by the server, which returns results to the client. Network traffic is minimized because only the requests and the results are shipped over the network.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn more about distributed databases

## Overview of Multitier Architecture

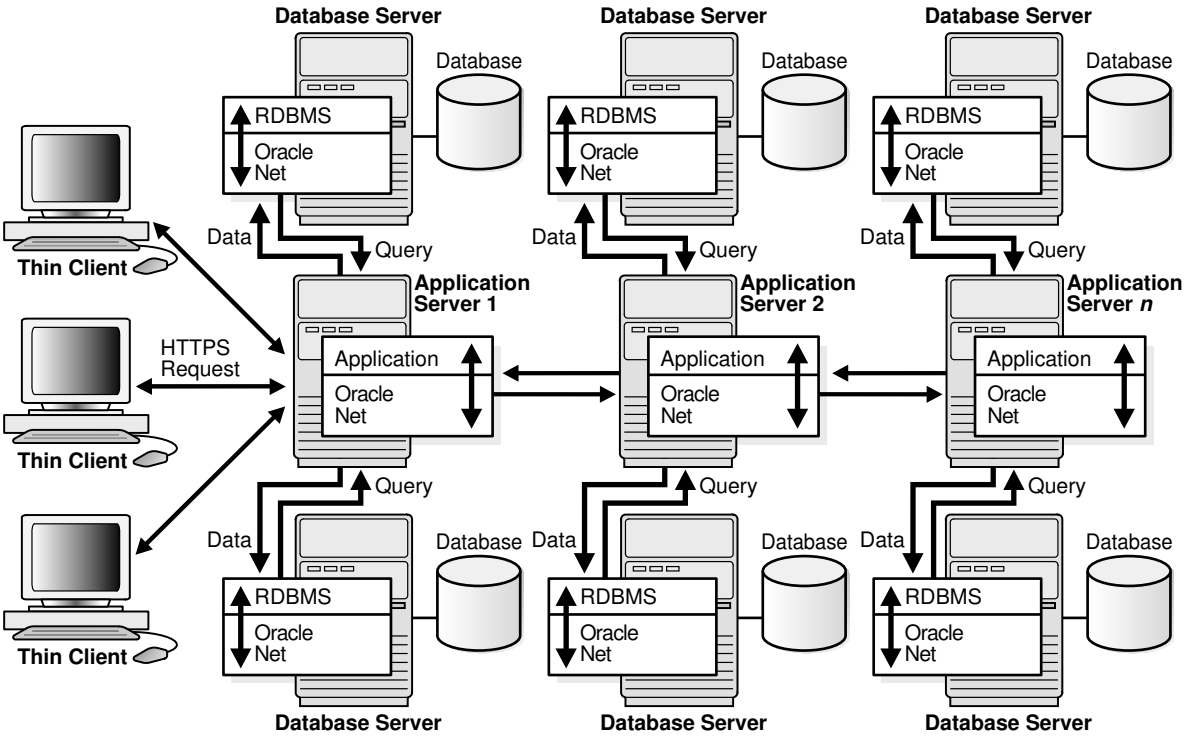
In a traditional multitier architecture, an application server provides data for clients and serves as an interface between clients and database servers.

This architecture enables use of an application server to:

- Validate the credentials of a client, such as a Web browser
- Connect to a database server
- Perform the requested operation

An example of a multitier architecture appears in Figure 18-3.

**Figure 18-3 A Multitier Architecture Environment**



- Clients
  A client initiates a request for an operation to be performed on the database server.
- Application Servers
  An application server provides access to the data for the client. It serves as an interface between the client and one or more database servers, and hosts the applications.
- Database Servers
  A database server provides the data requested by an application server on behalf of a client. The database performs the query processing.
- Service-Oriented Architecture (SOA)
  The database can serve as a Web service provider in traditional multitier or **service-oriented architecture (SOA)** environments.

## Clients

A client initiates a request for an operation to be performed on the database server.

The client can be a Web browser or other end-user program. In a multitier architecture, the client connects to the database server through one or more application servers.

## Application Servers

An application server provides access to the data for the client. It serves as an interface between the client and one or more database servers, and hosts the applications.

An application server permits thin clients, which are clients equipped with minimal software configurations, to access applications without requiring ongoing maintenance of the client computers. The application server can also perform data reformatting for the client, reducing the load on the client workstation.

The application server assumes the identity of the client when it is performing operations on the database server for this client. The best practice is to restrict the privileges of the application server to prevent it from performing unneeded and unwanted operations during a client operation.

## Database Servers

A database server provides the data requested by an application server on behalf of a client. The database performs the query processing.

The database server can audit operations performed by the application server on behalf of clients and on its own behalf. For example, a client operation can request information to display on the client, while an application server operation can request a connection to the database server.

In unified auditing, the database can append application contexts, which are application-specific name-value pairs, to records in the unified audit trail. You can configure which application contexts the database writes to database audit records.

> **See Also:**
>
> *Oracle Database Security Guide* for an introduction to auditing

## Service-Oriented Architecture (SOA)

The database can serve as a Web service provider in traditional multitier or **service-oriented architecture (SOA)** environments.

SOA is a multitier architecture relying on services that support computer-to-computer interaction over a network. In the context of SOA, a service is a self-sufficient functional endpoint that has a well defined functionality and service level agreement, can be monitored and managed, and can help enforce policy compliance.

SOA services are usually implemented as Web services accessible through the HTTP protocol. They are based on XML standards such as WSDL and SOAP.

The Oracle Database Web service capability, which is implemented as part of Oracle XML DB, must be specifically enabled by the DBA. Applications can then accomplish the following through database Web services:

- Submit SQL or XQuery queries and receive results as XML
- Invoke standalone PL/SQL functions and receive results
- Invoke PL/SQL package functions and receive results

Database Web services provide a simple way to add Web services to an application environment without the need for an application server. However, invoking Web services through application servers such as Oracle Fusion Middleware offers security, scalability, UDDI registration, and reliable messaging in an SOA environment. However, because database Web services integrate easily with Oracle Fusion Middleware, they may be appropriate for optimizing SOA solutions.

**See Also:**

- "PL/SQL Subprograms "
- *Oracle XML Developer's Kit Programmer's Guide* for information on enabling and using database Web services
- Oracle Fusion Middleware documentation for more information on SOA and Web services

# Overview of Grid Architecture

In an Oracle Database environment, **grid computing** is a computing architecture that effectively pools large numbers of servers and storage into a flexible, on-demand computing resource.

Modular hardware and software components can be connected and rejoined on demand to meet the changing needs of businesses.

> **See Also:**
>
> *Oracle Clusterware Administration and Deployment Guide* for more detailed information about server and storage grids

# Overview of Oracle Net Services Architecture

**Oracle Net Services** is a suite of networking components that provides enterprise-wide connectivity solutions in distributed, heterogeneous computing environments.

Oracle Net Services enables a network session from an application to a database instance and a database instance to another database instance.

Oracle Net Services provides location transparency, centralized configuration and management, and quick installation and configuration. It also lets you maximize system resources and improve performance. The shared server architecture increases the scalability of applications and the number of clients simultaneously connected to the database. The Virtual Interface (VI) protocol places most of the messaging burden on high-speed network hardware, freeing the CPU.

Oracle Net Services uses the communication protocols or application programmatic interfaces (APIs) supported by a wide range of networks to provide distributed database and distributed processing. After a network session is established, Oracle Net Services acts as a data courier for the client application and the database server, establishing and maintaining a connection and exchanging messages. Oracle Net Services can perform these tasks because it exists on each computer in the network.

This section contains the following topics:

- How Oracle Net Services Works
- The Oracle Net Listener
- Dedicated Server Architecture
- Shared Server Architecture
- Database Resident Connection Pooling

- How Oracle Net Services Works
  Oracle Database protocols accept SQL statements from the interface of the Oracle applications, and then package them for transmission to Oracle Database.
- The Oracle Net Listener
  The **Oracle Net Listener** (the listener) is a server-side process that listens for incoming client connection requests and manages traffic to the database. When a database instance starts, and at various times during its life, the instance contacts a listener and establishes a communication pathway to this instance.
- Dedicated Server Architecture
  In a **dedicated server** architecture, the server process created on behalf of each client process is called a dedicated server process (or *shadow process*).
- Shared Server Architecture
  In a **shared server** architecture, a dispatcher directs multiple incoming network session requests to a pool of shared server processes

- Database Resident Connection Pooling
  Database Resident Connection Pooling (DRCP) provides a connection pool of dedicated servers for typical Web application scenarios.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide* for an overview of Oracle Net architecture

## How Oracle Net Services Works

Oracle Database protocols accept SQL statements from the interface of the Oracle applications, and then package them for transmission to Oracle Database.

Transmission occurs through a supported industry-standard higher level protocol or API. Replies from Oracle Database are packaged through the same higher level communications mechanism. This work occurs independently of the network operating system.

Depending on the operating system that runs Oracle Database, the Oracle Net Services software of the database server could include the driver software and start an additional background process.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide* for more information about how Oracle Net Services works

## The Oracle Net Listener

The **Oracle Net Listener** (the listener) is a server-side process that listens for incoming client connection requests and manages traffic to the database. When a database instance starts, and at various times during its life, the instance contacts a listener and establishes a communication pathway to this instance.

Service registration enables the listener to determine whether a database service and its service handlers are available. A service handler is a dedicated server process or dispatcher that acts as a connection point to a database. During registration, the LREG process provides the listener with the instance name, database service names, and the type and addresses of service handlers. This information enables the listener to start a service handler when a client request arrives.

The following graphic shows two databases, each on a separate host. The database environment is serviced by two listeners, each on a separate host. The LREG process running in each database instance communicates with both listeners to register the database.

**Figure 18-4 Two Listeners**

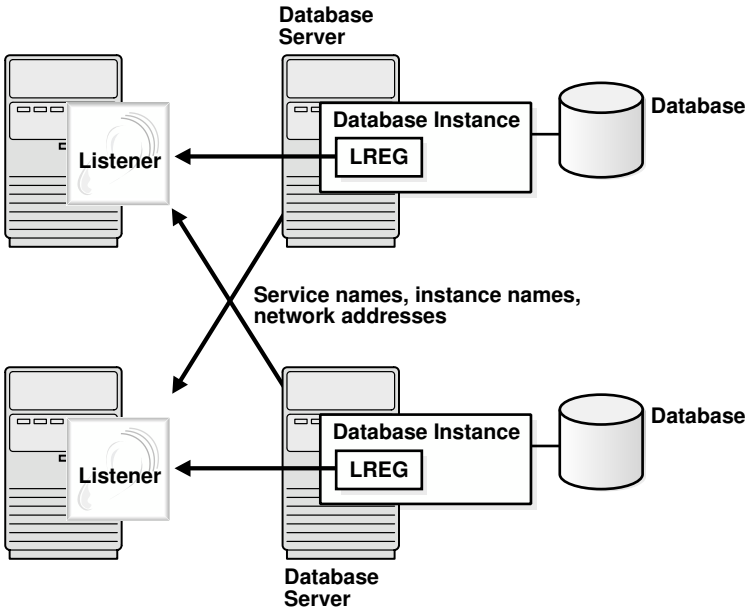
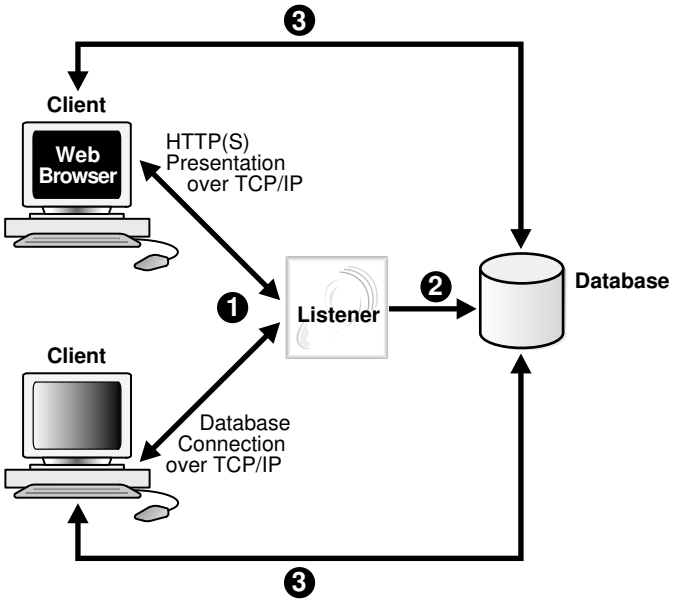



Figure 18-5 shows a browser making an HTTP connection and a client making a database connection through a listener. The listener does not need to reside on the database host.

**Figure 18-5 Listener Architecture**




The basic steps by which a client establishes a connection through a listener are:

1. A client process or another database requests a connection.
2. The listener selects an appropriate service handler to service the client request and forwards the request to the handler.

3. The client process connects directly to the service handler. The listener is no longer involved in the communication.

- Service Names
  A **service name** is a logical representation of a service used for client connections.
- Services in a Multitenant Environment
  Clients must connect to PDBs or application roots using services.
- Service Registration
  In Oracle Net, **service registration** is a feature by which the LREG process dynamically registers instance information with a listener.

> **See Also:**
>
> "Overview of Client Processes" and "Overview of Server Processes"

## Service Names

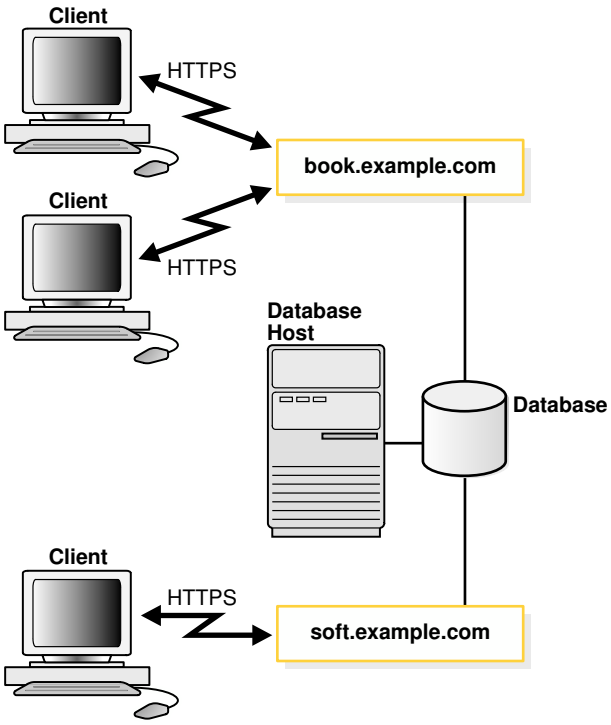
A **service name** is a logical representation of a service used for client connections.

When a client connects to a listener, it requests a connection to a service. When a database instance starts, it registers itself with a listener as providing one or more services by name. Thus, the listener acts as a mediator between the client and instances and routes the connection request to the right place.

A single service, as known by a listener, can identify one or more database instances. Also, a single database instance can register one or more services with a listener. Clients connecting to a service need not specify which instance they require.

Figure 18-6 shows one single-instance database associated with two services, `book.example.com` and `soft.example.com`. The services enable the same database to be identified differently by different clients. A database administrator can limit or reserve system resources, permitting better resource allocation to clients requesting one of these services.

**Figure 18-6 Multiple Services Associated with One Database**




> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide* to learn more about naming methods

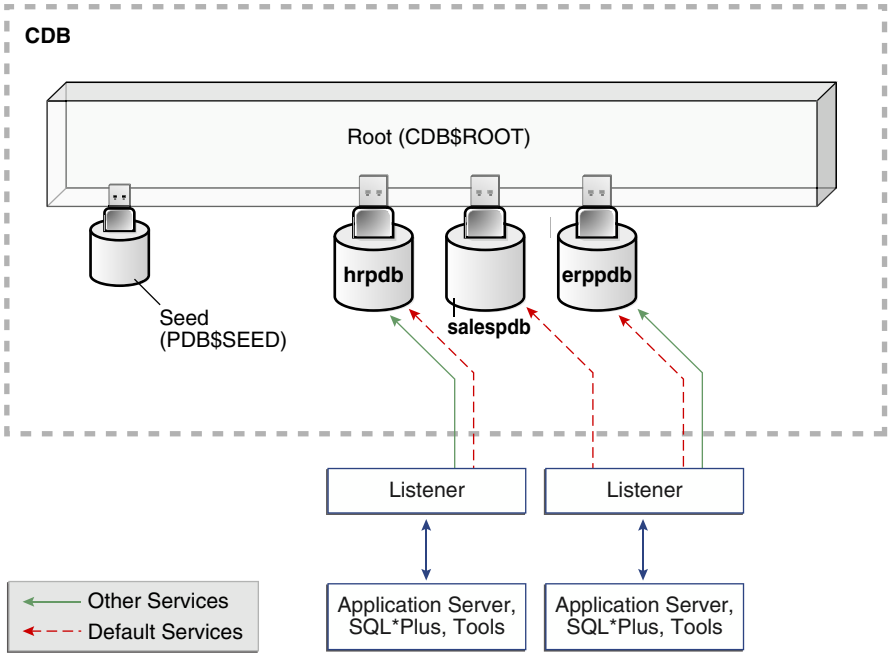
## Services in a Multitenant Environment

Clients must connect to PDBs or application roots using services.

A connection using a service name starts a new session in a PDB or application root. A foreground process, and therefore a session, at every moment of its lifetime, has a uniquely defined current container.

The following graphic shows two clients connecting to PDBs using two different listeners.

**Figure 18-7 Services in a CDB**




- Service Creation
  When you execute the `CREATE PLUGGABLE DATABASE` statement to create a PDB, the database automatically creates and starts a service inside the CDB.
- Connections to Containers in a CDB
  Typically, a CDB administrator must have appropriate privileges to provision PDBs and connect to various containers. CDB administrators are common users.

## Service Creation

When you execute the `CREATE PLUGGABLE DATABASE` statement to create a PDB, the database automatically creates and starts a service inside the CDB.

The default service has a property that identifies the PDB as the initial current container for the service. The property is shown in the `DBA_SERVICES.PDB` column.

- Default Services
  The default service has the same name as the PDB. The PDB name must be a valid service name, which must be unique within the CDB.
- Nondefault Services
  You can create additional services for each PDB, up to a per-CDB maximum of 10,000. Each additional service denotes its PDB as the initial current container.

## Default Services

The default service has the same name as the PDB. The PDB name must be a valid service name, which must be unique within the CDB.

When you create an application container, which requires specifying the `AS APPLICATION CONTAINER` clause, Oracle Database automatically creates a new default service for the application root. The service has the same name as the application container. Oracle Net Services must be configured properly for clients to access this service. Similarly, every

application PDB has its own default service name, and an application seed PDB has its own default service name.

**Example 18-1 Switching to a PDB Using a Default Service**

This example switches to the PDB names `salespdb` using the default service, which has the same name as the PDB:

```
ALTER SESSION SET CONTAINER = salespdb;
```

## Nondefault Services

You can create additional services for each PDB, up to a per-CDB maximum of 10,000. Each additional service denotes its PDB as the initial current container.

In Figure 18-7, nondefault services exist for `erppdb` and `hrpdb`. For example, in Figure 18-7 the PDB named `hrpdb` has a default service named `hrpdb`. The default service cannot be dropped.

When you switch to a container using `ALTER SESSION SET CONTAINER`, the session uses the default service for the container. Optionally, you can use a different service for the container by specifying `SERVICE = service_name`, where *`service_name`* is the name of the service. You might want to use a particular service so that the session can take advantage of its service attributes and features, such as service metrics, load balancing, Resource Manager settings, and so on.

**Example 18-2 Switching to a PDB Using a Nondefault Service**

In this example, the default service for `hrpdb` does not support all the service attributes and features such as service metrics, FAN, load balancing, Oracle Database Resource Manager, Transaction Guard, Application Continuity, and so on. You switch to a nondefault service as follows:

```
ALTER SESSION SET CONTAINER = hrpdb SERVICE = hrpdb_full;
```

## Connections to Containers in a CDB

Typically, a CDB administrator must have appropriate privileges to provision PDBs and connect to various containers. CDB administrators are common users.

The CDB administrator can use either of the following techniques:

- Connect directly to a PDB or application root.

  The user requires the `CREATE SESSION` privilege in the container.

- Use the `ALTER SESSION SET CONTAINER` statement, which is useful for both connection pooling and advanced CDB administration, to switch between containers. The syntax is `ALTER SESSION SET CONTAINER = container_name [SERVICE = service_name]`.

  For example, a CDB administrator can connect to the root in one session, and then in the same session switch to a PDB. In this case, the user requires the `SET CONTAINER` system privilege in the container.

The following table describes a scenario involving the CDB in Figure 18-7. Each row describes an action that occurs after the action in the preceding row. Common user `SYSTEM` queries the name of the current container and the names of PDBs in the CDB.

**Table 18-1 Services in a CDB**

| Operation | Description |
|---|---|
| `SQL> CONNECT SYSTEM@prod`<br>`Enter password: ********`<br>`Connected.` | The `SYSTEM` user, which is common to all containers in the CDB, connects to the root using service named `prod`. |
| `SQL> SHOW CON_NAME`<br><br>`CON_NAME`<br>`--------`<br>`CDB$ROOT` | `SYSTEM` uses the SQL*Plus command `SHOW CON_NAME` to list the name of the container to which the user is currently connected. `CDB$ROOT` is the name of the root container. |
| `SQL> SELECT NAME, PDB FROM V$SERVICES`<br>`  2  ORDER BY PDB, NAME;`<br><br>`NAME  PDB`<br>`----------  --------`<br>`SYS$BACKGROUND  CDB$ROOT`<br>`SYS$USERS  CDB$ROOT`<br>`prod.example.com  CDB$ROOT`<br>`erppdb.example.com  ERPPDB`<br>`erp.example.com  ERPPDB`<br>`hr.example.com  HRPDB`<br>`hrpdb.example.com  HRPDB`<br>`salespdb.example.com  SALESPDB`<br><br>`8 rows selected.` | A query of `V$SERVICES` shows that three PDBs exist with service names that match the PDB name. Both `hrpdb` and `erppdb` have an additional service. |
| `SQL> ALTER SESSION SET CONTAINER = hrpdb;`<br><br>`Session altered.` | `SYSTEM` uses `ALTER SESSION` to connect to `hrpdb`. |
| `SQL> SELECT SYS_CONTEXT`<br>`  2  ('USERENV', 'CON_NAME')`<br>`  3  AS CUR_CONTAINER FROM DUAL;`<br><br>`CUR_CONTAINER`<br>`-------------`<br>`HRPDB` | A query confirms that the current container is now `hrpdb`. |

> **See Also:**
>
> *Oracle Database SQL Language Reference* for the syntax and semantics of `ALTER SESSION SET CONTAINER`

## Service Registration

In Oracle Net, **service registration** is a feature by which the LREG process dynamically registers instance information with a listener.

This information enables the listener to forward client connection requests to the appropriate service handler. LREG provides the listener with information about the following:

- Names of the database services provided by the database
- Name of the database instance associated with the services and its current and maximum load
- Service handlers (dispatchers and dedicated servers) available for the instance, including their type, protocol addresses, and current and maximum load

Service registration is dynamic and does not require configuration in the `listener.ora` file. Dynamic registration reduces administrative overhead for multiple databases or instances.

The initialization parameter `SERVICE_NAMES` lists the services an instance belongs to. On startup, each instance registers with the listeners of other instances belonging to the same services. During database operations, the instances of each service pass information about CPU use and current connection counts to all listeners in the same services. This communication enables dynamic load balancing and connection failover.

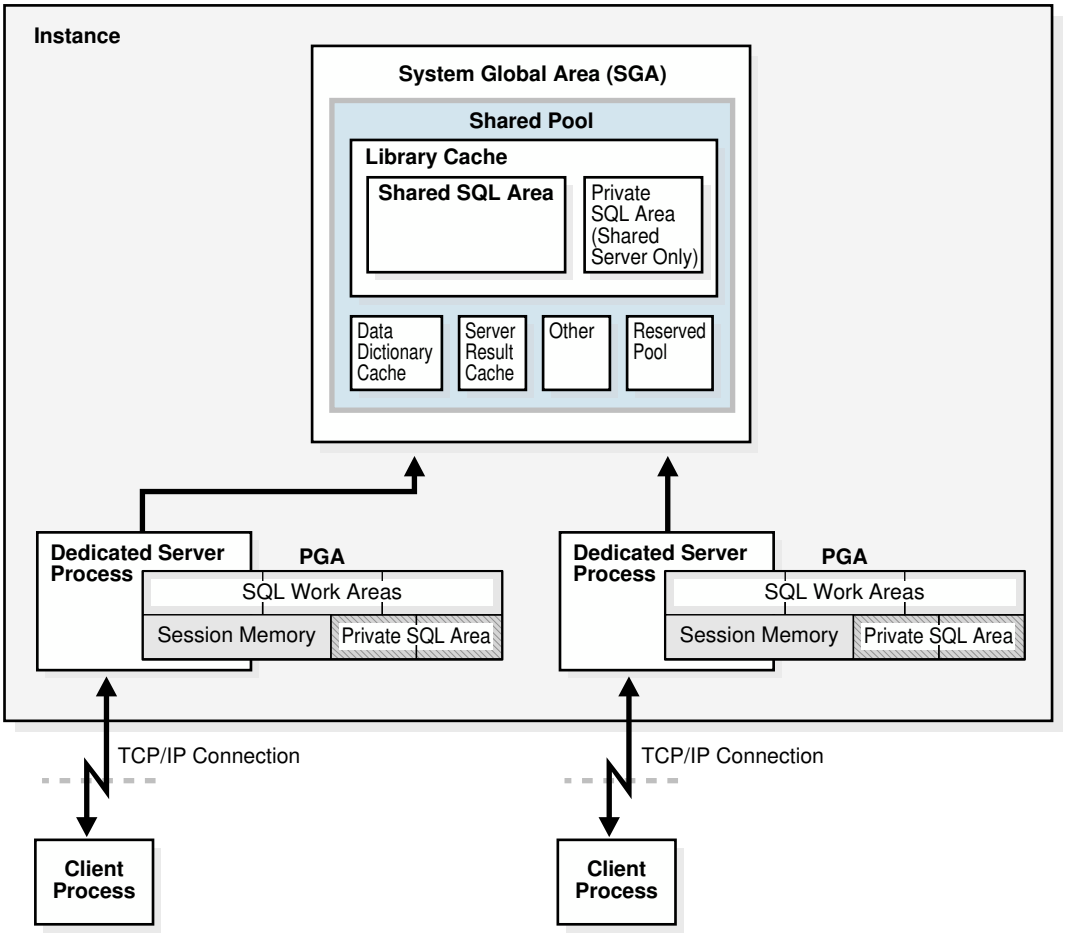
> **See Also:**
>
> - "Listener Registration Process (LREG)"
> - *Oracle Database Net Services Administrator's Guide* to learn more about service registration
> - *Oracle Real Application Clusters Administration and Deployment Guide* to learn about instance registration and client/service connections in Oracle RAC

# Dedicated Server Architecture

In a **dedicated server** architecture, the server process created on behalf of each client process is called a dedicated server process (or *shadow process*).

A dedicated server process is separate from the client process and acts only on its behalf, as shown in Figure 18-8.

**Figure 18-8 Oracle Database Using Dedicated Server Processes**




A one-to-one ratio exists between the client processes and server processes. Even when the user is not actively making a database request, the dedicated server process remains—although it is inactive and can be paged out on some operating systems.

Figure 18-8 shows user and server processes running on networked computers. However, the dedicated server architecture is also used if the same computer runs both the client application and the database code but the host operating system could not maintain the separation of the two programs if they were run in a single process. Linux is an example of such an operating system.

In the dedicated server architecture, the user and server processes communicate using different mechanisms:

- If the client process and the dedicated server process run on the same computer, then the program interface uses the host operating system's interprocess communication mechanism to perform its job.
- If the client process and the dedicated server process run on different computers, then the program interface provides the communication mechanisms (such as the network software and Oracle Net Services) between the programs.

Underutilized dedicated servers sometimes result in inefficient use of operating system resources. Consider an order entry system with dedicated server processes. A customer places an order as a clerk enters the order into the database. For most of the transaction, the clerk is talking to the customer while the server process dedicated to the clerk's client process

is idle. The server process is not needed during most of the transaction, and the system may be slower for other clerks entering orders if the system is managing too many processes. For applications of this type, the shared server architecture may be preferable.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide* to learn more about dedicated server processes

## Shared Server Architecture

In a **shared server** architecture, a dispatcher directs multiple incoming network session requests to a pool of shared server processes

The shared pool eliminates the need for a dedicated server process for each connection. An idle shared server process from the pool picks up a request from a common queue.

The potential benefits of shared server are as follows:

- Reduces the number of processes on the operating system

  A small number of shared servers can perform the same amount of processing as many dedicated servers.
- Reduces instance PGA memory

  Every dedicated or shared server has a PGA. Fewer server processes means fewer PGAs and less process management.
- Increases application scalability and the number of clients that can simultaneously connect to the database
- May be faster than dedicated server when the rate of client connections and disconnections is high

Shared server has several disadvantages, including slower response time in some cases, incomplete feature support, and increased complexity for setup and tuning. As a general guideline, only use shared server when you have more concurrent connections to the database than the operating system can handle.

The following processes are needed in a shared server architecture:

- A network listener that connects the client processes to dispatchers or dedicated servers (the listener is part of Oracle Net Services, not Oracle Database)

  > **Note:**
  >
  > To use shared servers, a client process must connect through Oracle Net Services, even if the process runs on the same computer as the Oracle Database instance.
- One or more dispatcher process (D*nnn*)
- One or more shared server processes

A database can support both shared server and dedicated server connections simultaneously. For example, one client can connect using a dedicated server while a different client connects to the same database using a shared server.

- Dispatcher Request and Response Queues
  A request from a user is a single API call that is part of the user's SQL statement.
- Restricted Operations of the Shared Server
  Specific administrative activities cannot be performed while connected to a dispatcher process, including shutting down or starting an instance and media recovery.

> **See Also:**
>
> - *Oracle Database Net Services Administrator's Guide* for more information about the shared server architecture
> - *Oracle Database Administrator's Guide* to learn how to configure a database for shared server

## Dispatcher Request and Response Queues

A request from a user is a single API call that is part of the user's SQL statement.

When a user makes a call, the following actions occur:

1. The dispatcher places the request on the request queue, where it is picked up by the next available shared server process.

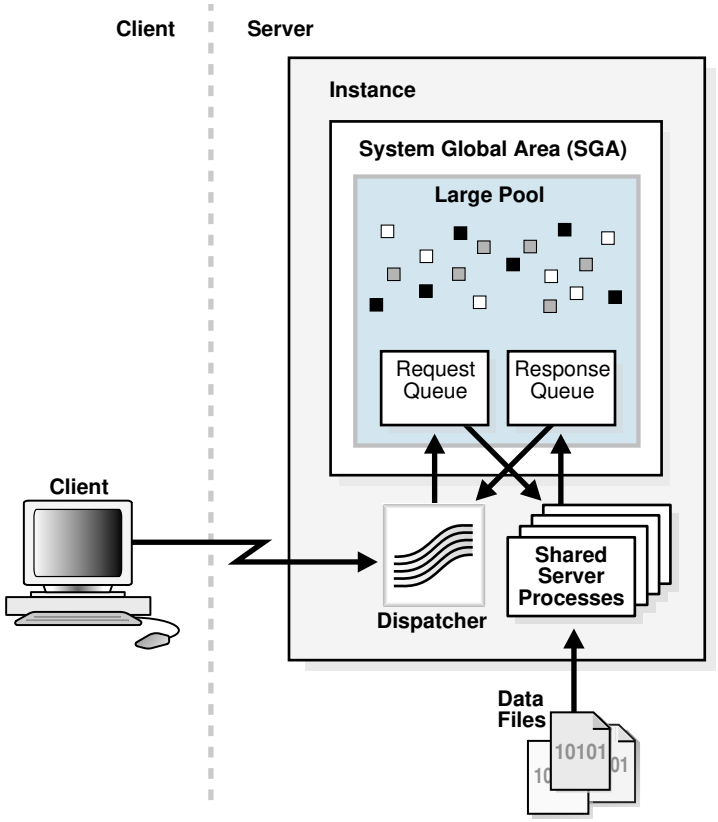
   The request queue is in the SGA and is common to all dispatcher processes of an instance.
2. The shared server processes check the common request queue for new requests, picking up new requests on a first-in-first-out basis.
3. One shared server process picks up one request in the queue and makes all necessary calls to the database to complete this request.

   A different server process can handle each database call. Therefore, requests to parse a query, fetch the first row, fetch the next row, and close the result set may each be processed by a different shared server.
4. When the server process completes the request, it places the response on the calling dispatcher's response queue. Each dispatcher has its own response queue.
5. The dispatcher returns the completed request to the appropriate client process.

For example, in an order entry system, each clerk's client process connects to a dispatcher. Each request made by the clerk is sent to this dispatcher, which places the request in the queue. The next available shared server picks up the request, services it, and puts the response in the response queue. When a request is completed, the clerk remains connected to the dispatcher, but the shared server that processed the request is released and available for other requests. While one clerk talks to a customer, another clerk can use the same shared server process.

Figure 18-9 shows how client processes communicate with the dispatcher across the API and how the dispatcher communicates user requests to shared server processes.

**Figure 18-9 The Shared Server Configuration and Processes**




- Dispatcher Processes (Dnnn)
  The **dispatcher processes** enable client processes to share a limited number of server processes.
- Shared Server Processes (Snnn)
  Each shared server process serves multiple client requests in the shared server configuration.

> **See Also:**
>
> "Large Pool"

## Dispatcher Processes (D*nnn*)

The **dispatcher processes** enable client processes to share a limited number of server processes.

You can create multiple dispatcher processes for a single database instance. The optimum number of dispatcher processes depending on the operating system limitation and the number of connections for each process.

> **Note:**
>
> Each client process that connects to a dispatcher must use Oracle Net Services, even if both processes run on the same host.

Dispatcher processes establish communication as follows:

1. When an instance starts, the network listener process opens and establishes a communication pathway through which users connect to Oracle Database.
2. Each dispatcher process gives the listener process an address at which the dispatcher listens for connection requests.

   At least one dispatcher process must be configured and started for each network protocol that the database clients will use.
3. When a client process makes a connection request, the listener determines whether the client process should use a shared server process:
   - If the listener determines that a shared server process is required, then the listener returns the address of the dispatcher process that has the lightest load, and the client process connects to the dispatcher directly.
   - If the process cannot communicate with the dispatcher, or if the client process requests a dedicated server, then the listener creates a dedicated server process and establishes an appropriate connection.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide* to learn how to configure dispatchers

## Shared Server Processes (S*nnn*)

Each shared server process serves multiple client requests in the shared server configuration.

Shared and dedicated server processes provide the same functionality, except shared server processes are not associated with a specific client process. Instead, a shared server process serves any client request in the shared server configuration.

The PGA of a shared server process does not contain UGA data, which must be accessible to all shared server processes. The shared server PGA contains only process-specific data.

All session-related information is contained in the SGA. Each shared server process must be able to access all sessions' data spaces so that any server can handle requests from any session. Space is allocated in the SGA for each session's data space.

> **See Also:**
>
> "Overview of the Program Global Area (PGA)"

## Restricted Operations of the Shared Server

Specific administrative activities cannot be performed while connected to a dispatcher process, including shutting down or starting an instance and media recovery.

These activities are typically performed when connected with administrator privileges. To connect with administrator privileges in a system configured with shared servers, you must specify that you are using a dedicated server process.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide* for the proper connect string syntax

# Database Resident Connection Pooling

Database Resident Connection Pooling (DRCP) provides a connection pool of dedicated servers for typical Web application scenarios.

A Web application typically makes a database connection, uses the connection briefly, and then releases it. Through DRCP, the database can scale to tens of thousands of simultaneous connections.

DRCP provides the following advantages:

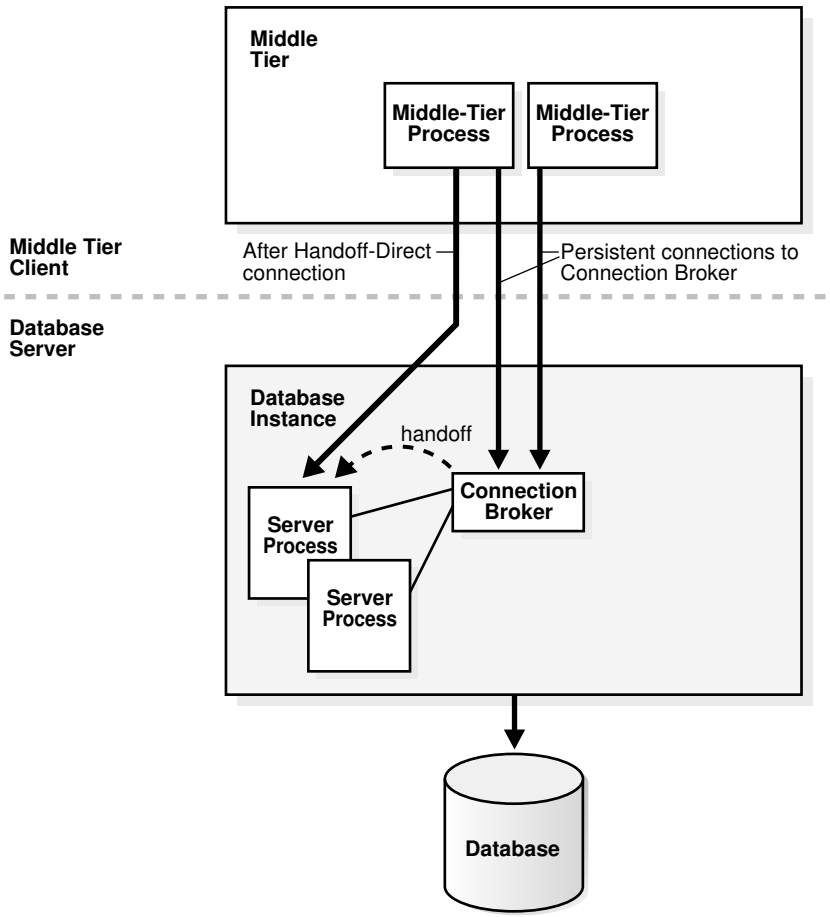
- Complements middle-tier connection pools that share connections between threads in a middle-tier process.
- Enables database connections to be shared across multiple middle-tier processes. These middle-tier processes may belong to the same or different middle-tier host.
- Enables a significant reduction in key database resources required to support many client connections. For example, DRCP reduces the memory required for the database and boosts the scalability of the database and middle tier. The pool of available servers also reduces the cost of re-creating client connections.
- Provides pooling for architectures with multi-process, single-threaded application servers, such as PHP and Apache, that cannot do middle-tier connection pooling.

DRCP uses a **pooled server**, which is the equivalent of a dedicated server process (not a shared server process) and a database session combined. The pooled server model avoids the overhead of dedicating a server for every connection that requires the server for a short period.

Clients obtaining connections from the database resident connection pool connect to an Oracle background process known as the connection broker. The connection broker implements the pool functionality and multiplexes pooled servers among inbound connections from client processes.

As shown in Figure 18-10, when a client requires database access, the connection broker picks up a server process from the pool and hands it off to the client. The client is directly connected to the server process until the request is served. After the server has finished, the server process is released into the pool. The connection from the client is restored to the broker.

**Figure 18-10 DRCP**




In DRCP, releasing resources leaves the session intact, but no longer associated with a connection (server process). Unlike in shared server, this session stores its UGA in the PGA, not in the SGA. A client can reestablish a connection transparently upon detecting activity.

> **See Also:**
>
> - "Connections and Sessions"
> - *Oracle Database Administrator's Guide* and *Oracle Call Interface Programmer's Guide* to learn more about DRCP

# Overview of the Program Interface

The **program interface** is the software layer between a database application and Oracle Database.

The program interface performs the following functions:

- Provides a security barrier, preventing destructive access to the SGA by client 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.

- Program Interface Structure
  The program interface consists of several different components.
- Program Interface Drivers
  A driver is a piece of software that transports data, usually across a network.
- Communications Software for the Operating System
  The lowest-level software connecting the user side to the Oracle Database side of the program interface is the communications software, which the host operating system provides.

## Program Interface Structure

The program interface consists of several different components.

These components include:

- Oracle call interface (OCI) or the Oracle run-time library (SQLLIB)
- The client or user side of the program interface
- Various Oracle Net Services drivers (protocol-specific communications software)
- Operating system communications software
- The server or Oracle Database side of the program interface (also called the OPI)

The user and Oracle Database sides of the program interface run Oracle software, as do the drivers.

## Program Interface Drivers

A driver is a piece of software that transports data, usually across a network.

Drivers perform operations such as connect, disconnect, signal errors, and test for errors. Drivers are specific to a communications protocol.

A default driver always exists. You can install multiple drivers, such as the asynchronous or DECnet drivers, and select one as the default driver, but allow a user to use other drivers by specifying a driver when connecting.

Different processes can use different drivers. A process can have concurrent connections to a single database or to multiple databases using different Oracle Net Services drivers.

**See Also:**

- Your system installation and configuration guide for details about choosing, installing, and adding drivers
- *Oracle Database Net Services Administrator's Guide* to learn about JDBC drivers

## Communications Software for the Operating System

The lowest-level software connecting the user side to the Oracle Database side of the program interface is the communications software, which the host operating system provides.

DECnet, TCP/IP, LU6.2, and ASYNC are examples. The communication software can be supplied by Oracle, but it is usually purchased separately from the hardware vendor or a third-party software supplier.

# Glossary

**access driver**

In the external table infrastructure, the API that interprets the external data for the database. The access driver runs inside the database, which uses the driver to read the data in the external table.

**access path**

The means by which data is retrieved from a database. For example, a query using an index and a query using a full table scan use different access paths.

**ACID properties**

The basic properties of a database transaction that all Oracle Database transactions must obey. ACID is an acronym for atomicity, consistency, isolation, and durability.

**active online redo log file**

An online redo log file that may contain data that is required for database instance recovery.

**active session**

A database session that is using CPU and is not waiting for an event in the idle wait class.

**Active Session History (ASH)**

A part of the database self-management framework that samples active database sessions each second, writing the data to memory and persistent storage.

**active transaction**

A transaction that has started but not yet committed or rolled back.

**adaptive query optimization**

A set of capabilities that enables the adaptive optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better optimizer statistics.

Adaptive optimization is helpful when existing statistics are not sufficient to generate an optimal plan.

**ADDM**

Automatic Database Diagnostic Monitor. An Oracle Database infrastructure that enables a database to diagnose its own performance and determine how identified problems could be resolved.

**ADR**

Automatic Diagnostic Repository. A file-based hierarchical data store for managing information, including network tracing and logging.

**ADR base**

The ADR root directory. The ADR base can contain multiple ADR homes, where each ADR home is the root directory for all diagnostic data—traces, dumps, the alert log, and so on—for an instance of an Oracle product or component.

**ADR home**

The root directory for all diagnostic data—traces, dumps, the alert log, and so on—for an instance of an Oracle product or component. For example, in an Oracle RAC environment with shared storage and Oracle ASM, each database instance and each Oracle ASM instance has its own ADR home.

**advanced index compression**

An extension and enhancement of prefix compression for supported unique and non-unique indexes on heap-organized tables. Unlike prefix compression, which uses fixed duplicate key elimination for every block, advanced compression uses adaptive duplicate key elimination on a per-block basis.

**advanced row compression**

A type of table compression, intended for OLTP applications, that compresses data manipulated by any SQL operation.
See also basic table compression.

**aggregate function**

A function such as `COUNT` that operates on a group of rows to return a single row as a result.

**alert log**

A file that provides a chronological log of database messages and errors. The alert log is stored in the ADR.

**analytic function**

A function that operates on a group of rows to return multiple rows as a result.

**analytic query**

A "what if" query that answers a business question. Typically, analytic queries involve joins and aggregation, and require scanning a very large amount of input data to produce a relatively small amount of output.

**analytic view**

A type of view that encapsulates aggregations, calculations, and joins of fact data. Analytic views organize data using a dimensional model. They allow you to easily add aggregations and calculations to data sets and to present data in views that can be queried with relatively simple SQL.

**antijoin**

A join that returns rows from the left side of the predicate for which there are no corresponding rows on the right side of the predicate.

**application**

Within an application root, an application is a named, versioned set of data and metadata created by a common user. An application might include an application common user, an application common object, or some multiple and combination of the preceding.

**application architecture**

The computing environment in which a database application connects to an Oracle database. The two most common database architectures are client/server and multitier.

**application common object**

A shared database object created while connected to an application root. The metadata (for a metadata-linked object) or data (for a data-linked common object) is shared by application PDBs in the application container.

**application common user**

A common user created while connected to an application root. The metadata (for a metadata-linked common object) or data (for a data-linked common object) is shared by application PDBs in the application container.

**application container**

A named set of application PDBs plugged in to an application root. An application container may contain an application seed.

**application context**

An attribute name-value pair in a specified namespace. Applications set various contexts before executing actions on the database.

**Application Continuity**

A feature that enables the replay, in a nondisruptive and rapid manner, of a request against the database after a recoverable error that makes the database session unavailable.

**application domain index**

A customized index specific to an application.

**application PDB**

A PDB that is plugged in to an application container.

**application root**

The root container within an application container. Every application container has exactly one application root. An application root shares some characteristics with the CDB root, because it can contain common objects, and some characteristics with a PDB, because it is created with the `CREATE PLUGGABLE DATABASE` statement.

**application seed**

An optional application PDB that serves as a template for creating other PDBs within an application container. An application container includes 0 or 1 application seed.

**application server**

Software that provides an interface between the client and one or more database servers, and hosts the applications.

**archive compression**

Hybrid Columnar Compression specified with `COLUMN STORE COMPRESS FOR ARCHIVE`. This type uses higher compression ratios than `COLUMN STORE COMPRESS FOR QUERY`, and is useful for compressing data that will be stored for long periods of time.

**archived redo log file**

A member of the online redo log that has been archived by Oracle Database. The archived redo log files can be applied to a database backup in media recovery.

**ARCHIVELOG mode**

A mode of the database that enables the archiving of the online redo log.

**archiver process (ARCn)**

The background process that archives online redo log files.

**archiving**

The operation of generating an archived redo log file.

**ascending index**

An index in which data is stored in ascending order. By default, character data is ordered by the binary values contained in each byte of the value, numeric data from smallest to largest number, and date from earliest to latest value.

**attribute-clustered table**

A heap-organized table that stores data in close proximity on disk based on user-specified clustering directives.

**audit trail**

A location that stores audit records.

**Automatic Database Diagnostic Monitor (ADDM)**

See ADDM.

**Automatic Diagnostic Repository (ADR)**

See ADR.

**automatic memory management**

The mode in which Oracle Database manages the SGA and instance PGA memory completely automatically.

**automatic segment space management (ASSM)**

A method of storage space management that uses bitmaps to manage segment space instead of free lists.

**automatic undo management mode**

A mode of the database in which it automatically manages undo space in a dedicated undo tablespace.

See also manual undo management mode.

**Automatic Workload Repository (AWR)**

See AWR.

**autonomous transaction**

A independent transaction that can be called from another transaction, called the main transaction.

**AWR**

Automatic Workload Repository (AWR). A built-in repository in every Oracle database. Oracle Database periodically makes a snapshot of its vital statistics and workload information and stores them in AWR.

**AWR baseline**

A collection of statistic rates usually taken over a period when the system is performing well at peak load

**AWR snapshot**

A set of performance statistics captured in AWR at a specific time.

**B-tree index**

An index organized like an upside-down tree. A B-tree index has two types of blocks: branch blocks for searching and leaf blocks that store values. The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. The "B" stands for "balanced" because all leaf blocks automatically stay at the same depth.

**background process**

A process that consolidates functions that would otherwise be handled by multiple Oracle programs running for each client process. The background processes asynchronously perform I/O and monitor other Oracle processes.

See also database instance; Oracle process.

**backup**

A copy of data. A backup can include crucial parts of the database such as data files, the server parameter file, and control file.

**backup piece**

The smallest unit of a backup set.

**backup set**

A proprietary RMAN backup format that contains data from one or more data files, archived redo log files, or control files or server parameter file.

**basic table compression**

A type of table compression intended for bulk load operations. You must use direct path `INSERT` operations, `ALTER TABLE . . . MOVE` operations, or online table redefinition to achieve basic table compression.

**big table cache**

An optional, integrated portion of the database buffer cache that uses a temperature-based, object-level replacement algorithm instead of the traditional LRU-based, block-level replacement algorithm.

**bigfile tablespace**

A tablespace that contains one very large data file or temp file.

**bind variable**

A placeholder in a SQL statement that must be replaced with a valid value or value address for the statement to execute successfully. By using bind variables, you can write a SQL statement that accepts inputs or parameters at run time. The following example shows a query that uses `v_empid` as a bind variable:

```
SELECT * FROM employees WHERE employee_id = :v_empid;
```

**bitmap index**

A database index in which the database stores a bitmap for each index key instead of a list of rowids.

**bitmap join index**

A bitmap index for the join of two or more tables.

**bitmap merge**

An operation that merges bitmaps retrieved from bitmap index scans. For example, if the `gender` and `DOB` columns have bitmap indexes, then the database may use a bitmap merge if the query predicate is `WHERE gender='F' AND DOB > 1966`.

**block corruption**

A data block that is not in a recognized Oracle format, or whose contents are not internally consistent.

**block header**

A part of a data block that includes information about the type of block, the address of the block, and sometimes transaction information.

**block overhead**

Space in a data block that stores metadata required for managing the block. The overhead includes the block header, table directory, and row directory.

**branch block**

In a B-tree index, a block that the database uses for searching. The leaf blocks store the index entries. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

**B-tree index**

An index organized like an upside-down tree. A B-tree index has two types of blocks: branch blocks for searching and leaf blocks that store values. The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. The "B" stands for "balanced" because all leaf blocks automatically stay at the same depth.

**buffer**

A main memory address in the database buffer cache. A buffer caches currently and recently used data blocks read from disk. When a new block is needed, the database can replace an old data block with a new one.

**buffer cache hit ratio**

The measure of how often the database found a requested block in the buffer cache without needing to read it from disk.

**buffer header**

A memory structure that stores metadata about a buffer.

**buffer pool**

A collection of buffers in the SGA.

**business intelligence**

The analysis of an organization's information as an aid to making business decisions.

**byte semantics**

Treatment of strings as a sequence of bytes. Offsets into strings and string lengths are expressed in bytes.

**cache recovery**

The automatic phase of instance recovery where Oracle Database applies all committed and uncommitted changes in the online redo log files to the affected data blocks.

**cardinality**

The ratio of distinct values to the number of table rows. A column with only two distinct values in a million-row table would have low cardinality.

**Cartesian join**

A join in which one or more of the tables does not have any join conditions to any other tables in the statement. The optimizer joins every row from one data source with every row from the other data source, creating the Cartesian product of the two sets.

**CDB**

An Oracle Database installation that contains at least one PDB. Starting in Oracle Database 21c, every Oracle database is a CDB.

**CDB administrator**

A database administrator who manages a CDB. A PDB administrator manages individual PDBs within the CDB.

**CDB root**

In a multitenant container database (CDB), a collection of schemas, schema objects, and nonschema objects to which all PDBs belong. Every CDB has exactly one root container, which stores the system metadata required to manage PDBs. All PDBs belong to the CDB root.

**character encoding**

A code that pairs each character from a given repertoire with a code unit to facilitate data storage.

**character semantics**

Treatment of strings as a sequence of characters. Offsets into strings and string lengths are expressed in characters (character codes).

**character set**

An encoding scheme used to display characters on your computer screen.

**check constraint**

A constraint on a column or set of columns that requires a specified condition to be true or unknown for every row.

**checkpoint**

1. A data structure that marks the checkpoint position, which is the SCN in the redo thread where instance recovery must begin. Checkpoints are recorded in the control file and each data file header, and are a crucial element of recovery.

2. The writing of dirty data blocks in the database buffer cache to disk. The database writer (DBW) process writes blocks to disk to synchronize the buffer cache with the data files.

**checkpoint process (CKPT)**

The background process that updates the control file and data file headers with checkpoint information and signals DBW to write blocks to disk.

**clean restore point**

A PDB restore point that is created when the PDB is closed. A Flashback PDB to a clean restore point does not require restoring backups or creating a temporary instance.

**child cursor**

The cursor containing the plan, compilation environment, and other information for a statement whose text is stored in a parent cursor. The parent cursor is number `0`, the first child is number `1`, and so on. Child cursors reference exactly the same SQL text as the parent cursor, but are

different. For example, two statements with the text `SELECT * FROM mytable` use different cursors when they reference tables named `mytable` in different schemas.

**circular reuse record**

A type of control file record that contains noncritical information that is eligible to be overwritten if needed. When all available record slots are full, the database either expands the control file to make room for a new record or overwrites the oldest record.

**client**

In client/server architecture, the front-end database application that interacts with a user. The client portion has no data access responsibilities.

**client character set**

The character set for data entered or displayed by a client application. The character set for the client and database can be different.

**client process**

A process that executes the application or Oracle tool code. When users run client applications such as SQL*Plus, the operating system creates client processes to run the applications.

See also Oracle process.

**client/server architecture**

Software architecture based on a separation of processing between two CPUs, one acting as the client in the transaction, requesting and receiving services, and the other as the server that provides services in a transaction.

**cluster file system**

A distributed file system that is a cluster of servers that collaborate to provide consistency and high performance to their clients.

**cluster index**

A B-tree index on the cluster key.

**cluster key**

In a table cluster, the column or columns that the clustered tables have in common. For example, the `employees` and `departments` tables share the `department_id` column. Specify the cluster key when creating the table cluster and when creating every table added to the table cluster.

**cold buffer**

A buffer in the database buffer cache that has not been recently used.

**column**

Vertical space in a table that represents a domain of data. A table definition includes a table name and set of columns. Each column has a name and data type.

**columnar format**

The column-based format for objects that reside in the In-Memory Column Store. The columnar format contrasts with the row format that the database uses to store objects in the database buffer cache and in data files.

**commit**

Action that ends a database transaction and makes permanent all changes performed in the transaction.

**commit cleanout**

The automatic removal of lock-related transaction information (ITL entry) from the blocks after a commit. The database removes the ITL entry only if modified blocks containing data from the committed transaction are still in the SGA, and if no other session is modifying them.

**common object**

An object that resides either in the CDB root or an application root that shares either data (a data-linked common object) or metadata (a metadata-linked common object). All common objects in the CDB root are Oracle-supplied. A common object in an application root is called an application common object.

**common role**

A role that exists in all containers in a multitenant container database (CDB).

**common user**

In a multitenant container database (CDB), a database user that exists with the same identity in multiple containers. A common user created in the CDB root has the same identity in every existing and future PDB. A common user created in an application container has the same identity in every existing and future application PDB in this application container.

**complete refresh**

An execution of the query that defines a materialized view. A complete refresh occurs when you initially create the materialized view, unless the materialized view references a prebuilt table, or you define the table as `BUILD DEFERRED`.

**composite database operation**

Activity between two points in time in a single database session.

**composite index**

An index on multiple columns in a table.

**composite partitioning**

A partitioning strategy in which a table is partitioned by one data distribution method and then each partition is further divided into subpartitions using a second data distribution method.

**composite unique key**

A set of two or more columns with a unique key constraint.

**compound trigger**

A trigger can that can fire at multiple timing points. For example, a compound trigger might fire both before and after the triggering statement.

**compression unit**

In Hybrid Columnar Compression, a logical construct that stores a set of rows. When you load data into a table, the database stores groups of rows in columnar format, with the values for each column stored and compressed together. After the database has compressed the column data for a set of rows, the database fits the data into the compression unit.

**concatenated index**

See composite index.

**condition**

The combination of one or more expressions and logical operators in a SQL statement that returns a value of `TRUE`, `FALSE`, or `UNKNOWN`. For example, the condition `1=1` always evaluates to `TRUE`.

**conflicting write**

In a read committed transaction, a situation that occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction.

**connection**

Communication pathway between a client process and an Oracle database instance.

See also session.

**connection pooling**

A resource utilization and user scalability feature that maximizes the number of sessions over a limited number of protocol connections to a shared server.

**consistent backup**

A whole database backup that you can open with the `RESETLOGS` option without performing media recovery. By its nature, a consistent backup of the whole database does not require the application of redo to be made consistent.

See also inconsistent backup.

**consistent read get**

The retrieval of a version of a block in the database buffer cache that is consistent to a specific SCN (part of read consistency). If the database needs a block to satisfy a query, and if no block in the database buffer cache is consistent to the correct SCN, then the database attempts to obtain the correct version of the block from undo data.

**container**

In a multitenant container database (CDB), either the root or a PDB.

**context**

A set of application-defined attributes that validates and secures an application. The SQL statement `CREATE CONTEXT` creates namespaces for contexts.

**control file**

A binary file that records the physical structure of a database and contains the names and locations of redo log files, the time stamp of the database creation, the current log sequence number, checkpoint information, and so on.

**cube**

An organization of measures with identical dimensions and other shared characteristics. The edges of the cube contain the dimension members, whereas the body of the cube contains the data values.

**current mode get**

The retrieval of the version of a data block as it exists right now in the buffer cache, without using read consistency. Only one version of a block exists in current mode at any one time.

**current online redo log file**

The online redo log file to which the log writer (LGWR) process is actively writing.

**cursor**

A handle or name for a private SQL area in the PGA. Because cursors are closely associated with private SQL areas, the terms are sometimes used interchangeably.

**data block**

Smallest logical unit of data storage in Oracle Database. Other names for data blocks include Oracle blocks or pages. One data block corresponds to a specific number of bytes of physical space on disk.

See also extent; segment.

**data concurrency**

Simultaneous access of the same data by many users. A multiuser database management system must provide adequate concurrency controls so that data cannot be updated or changed improperly, compromising data integrity.

See also data consistency.

**data consistency**

A consistent view of the data by each user in a multiuser database.

See also data concurrency.

**data corruption**

An error that occurs when a hardware, software, or network component causes corrupt data to be read or written.

**data dictionary**

A read-only collection of database tables and views containing reference information about the database, its structures, and its users.

**data dictionary (DDL) lock**

A lock that protects the definition of a schema object while an ongoing DDL operation acts on or refers to the object. Oracle Database acquires a DDL lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks.

**data dictionary cache**

A memory area in the shared pool that holds data dictionary information. The data dictionary cache is also known as the *row cache* because it holds data as rows instead of buffers, which hold entire data blocks.

**data dictionary view**

A predefined view of tables or other views in the data dictionary. Data dictionary views begin with the prefix `DBA_`, `ALL_`, or `USER_`.

**data file**

A physical file on disk that was created by Oracle Database and contains the data for a database. The data files can be located either in an operating system file system or Oracle ASM disk group.

**data link**

In a PDB, an internal mechanism that points to data (not metadata) in the root. For example, AWR data resides in the root. Each PDB uses an object link to point to the AWR data in the root, thereby making views such as `DBA_HIST_ACTIVE_SESS_HISTORY` and `DBA_HIST_BASELINE` accessible in each separate container.

**data-linked common object**

A common object that exists either in the CDB root or an application root. The data, rather than the metadata, is shared by any PDB that contains a data link that points to the common object.

**data integrity**

Business rules that dictate the standards for acceptable data. These rules are applied to a database by using integrity constraints and triggers to prevent invalid data entry.

**data mining**

The automated search of large stores of data for patterns and trends that transcend simple analysis.

**Data Recovery Advisor**

An Oracle Database infrastructure that automatically diagnoses persistent data failures, presents repair options to the user, and executes repairs at the user's request.

**data segment**

The segment containing the data for a nonclustered table, table partition, or table cluster.

See also extent.

**data type**

In SQL, a fixed set of properties associated with a column value or constant. Examples include `VARCHAR2` and `NUMBER`. Oracle Database treats values of different data types differently.

**data warehouse**

A relational database designed for query and analysis rather than for OLTP.

**database access control**

Restricting data access and database activities. For example, the restriction of users from querying specified tables or executing specified database statements.

**database application**

A software program that interacts with a database to access and manipulate data.

**database authentication**

The process by which a user presents credentials to the database, which verifies the credentials and allows access to the database.

**database block size**

The data block size for a database set when it is created. The size is set for the `SYSTEM` and `SYSAUX` tablespaces and is the default for all other tablespaces. The database block size cannot be changed except by re-creating the database.

**database buffer cache**

The portion of the system global area (SGA) that holds copies of data blocks. All client processes concurrently connected to the database instance share access to the buffer cache.

**database character set**

A character encoding scheme that determines which languages can be represented in a database.

**database driver**

Software that sits between an application and an Oracle database. The driver translates the API calls made by the application into commands that the database can process. By using an ODBC driver, an application can access any data source, including data stored in spreadsheets. The ODBC driver performs all mappings between the ODBC standard and the database.

**database instance**

The combination of the system global area (SGA) and background processes. An instance is associated with one and only one database. Every database instance is either a read/write database instance or a read-only database instance. In an Oracle Real Application Clusters configuration, multiple instances access a single database.

**database link**

A schema object in one database that enables users to access objects in a different database.

**database management system (DBMS)**

Software that controls the storage, organization, and retrieval of data.

**database object**

An object in the database that can be manipulated with SQL. Schema objects such as tables and indexes reside in schemas. Nonschema objects such as directories and roles do not reside in schemas.

**database operation**

In the context of database monitoring, a logical entity that includes a SQL statement, a PL/SQL block, or a composite of the two.

**database point-in-time recovery**

A type of media recovery that results in a noncurrent version of the database. In this case, you do not apply all of the redo generated after the restored backup.

**database security**

The aspect of database administration that involves user authentication, encryption, access control, and monitoring.

**database server**

A server that reliably manages a large amount of data in a multiuser environment so that users can concurrently access the same data. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.

**Database Server Grid**

A collection of commodity servers connected together to run on one or more databases.

**database service**

A named representation of one or more database instances. The service name for an Oracle database is normally its global database name. Clients use the service name to connect to one or more database instances.

**Database Storage Grid**

A collection of low-cost modular storage arrays combined together and accessed by the computers in the Database Server Grid.

**database user**

An account through which you can log in to an Oracle database.

**database writer (DBW)**

A background process that writes buffers in the database buffer cache to data files.

**DDL**

Data definition language. Includes statements such as `CREATE TABLE` or `ALTER INDEX` that define or change a data structure.

**deadlock**

A situation in which two or more users are waiting for data locked by each other. Such deadlocks are rare in Oracle Database.

**declarative language**

A nonprocedural language that describes what should be done, now how to do it. SQL and Prolog are examples of declarative languages. SQL is declarative in the sense that users specify the result that they want, not how to derive it.

**dedicated server**

A database configuration in which a server process handles requests for a single client process.

See also shared server.

**deferrable constraint**

A constraint that permits a `SET CONSTRAINT` statement to defer constraint checking until a `COMMIT` statement is issued. A deferrable constraint enables you to disable the constraint temporarily while making changes that might violate the constraint.

**deferred insert**

An insert with the `MEMOPTIMIZE_WRITE` hint into a table specified as `MEMOPTIMIZE FOR WRITE`. Deferred inserts cannot be rolled back and do not use the consistency mechanisms of the database buffer cache. Background processes write deferred inserts to the data files asynchronously.

**definer's rights PL/SQL procedure**

A procedure that executes with the privileges of its owner, not its current user.

**degree of parallelism**

The number of parallel execution servers associated with a single operation. Parallel execution is designed to effectively use multiple CPUs. Oracle Database parallel execution framework enables you to either explicitly choose a specific degree of parallelism or to rely on Oracle Database to automatically control it.

**dependent object**

In a schema object dependency, the object whose definition references another object. For example, if the definition of object A references object B, then A is a dependent object on B.

**descending index**

An index in which data is stored on a specified column or columns in descending order.

**dimension**

A structure that categorizes data to enable users to answer business questions. Commonly used dimensions are customers, products, and time.

**dimension table**

A relational table that stores all or part of the values for a dimension in a star or snowflake schema. Dimension tables typically contain columns for the dimension keys, levels, and attributes.

**direct path INSERT**

An `INSERT` in which the database writes data directly to the data files, bypassing the database buffer cache. The database appends the inserted data to the existing data in the table.

**direct path read**

A single or multiblock read into the PGA, bypassing the SGA.

**directory object**

A database object that specifies an alias for a directory on the server file system where external binary file LOBs (BFILEs) and external table data are located. All directory objects are created in a single namespace and are not owned by an individual schema.

**dirty read**

The situation that occurs when a transaction reads uncommitted data written by another transaction. Oracle Database *never* permits dirty reads.

**dispatcher**

See dispatcher process (Dnnn).

**dispatcher process (D*nnn*)**

Optional background process present only when a shared server configuration is used. Each dispatcher process is responsible for routing requests from connected client processes to available shared server processes and returning the responses.

**distributed database**

A set of databases in a distributed system that can appear to applications as a single data source.

**distributed environment**

A network of disparate systems that seamlessly communicate with each other.

**distributed processing**

The operations that occur when an application distributes its tasks among different computers in a network.

**distributed transaction**

A transaction that includes statements that, individually or as a group, update data on nodes of a distributed database. Oracle Database ensures the integrity of data in distributed transactions using the two-phase commit mechanism.

**DML**

Data manipulation language. Includes statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.

**DML lock**

A lock that prevents destructive interference of simultaneous conflicting DML or DDL operations. DML statements automatically acquire row locks and table locks.

**dynamic performance view**

A special views that is continuously updated while a database is open and in use. The dynamic performance views are sometimes called *V$ views*.

**dynamic SQL**

SQL whose complete text is not known until run time. Dynamic SQL statements are stored in character strings that are entered into, or built by, the program at run time.

**edition**

A private environment in which you can redefine database objects. Edition-based redefinition enables you to upgrade an application's database objects while the application is in use, thus minimizing or eliminating downtime.

**encryption**

The process of transforming data into an unreadable format using a secret key and an encryption algorithm.

**equijoin**

A join with a join condition containing an equality operator.

**ETL**

Extraction, transformation, and loading (ETL). The process of extracting data from source systems and bringing it into a data warehouse.

**exclusive lock**

A lock that prevents the associated resource from being shared. The first transaction to obtain an exclusive lock on a resource is the only transaction that can alter the resource until the lock is released.

**executable SQL statement**

A SQL statement that generates calls to a database instance, including DML and DDL statements and the `SET TRANSACTION` statement.

**execution plan**

The combination of steps used by the database to execute a SQL statement. Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement. You can override execution plans by using a hint.

**expression**

A combination of one or more values, operators, and SQL functions that resolves to a value. For example, the expression `2*2` evaluates to `4`. In general, expressions assume the data type of their components.

**extent**

Multiple contiguous data blocks allocated for storing a specific type of information. A segment is made up of one or more extents.

See also data block.

**external table**

A read-only table whose metadata is stored in the database but whose data in stored in files outside the database. The database uses the metadata describing external tables to expose their data as if they were relational tables.

**extraction, transformation, and loading (ETL)**

See ETL.

**fact**

Data that represents a business measure, such as sales or cost data.

**fact table**

A table in a star schema of a data warehouse that contains factual data. A fact table typically has two types of columns: those that contain facts and those that are foreign keys to dimension tables.

**fast full index scan**

A full index scan in which the database reads all the blocks in the index using multiblock reads, and then discards the branch blocks, returning the index blocks in no particular order.

**fast recovery area**

An optional disk location that stores recovery-related files such as control file and online redo log copies, archived redo log files, flashback logs, and RMAN backups.

**fault tolerance**

The protection provided by a high availability architecture against the failure of a component in the architecture.

**field**

In a table, the intersection of a row and column.

**file system**

A data structure built inside a contiguous disk address space.

**fine-grained auditing**

A type of database auditing that enables you to audit specific table columns, and to associate event handlers during policy creation.

**fixed SGA**

An internal housekeeping area that contains a variety of information, including general information about the state of the database and the instance, and information communicated between processes.

**flashback data archive process (FBDA)**

The background process that archives historical rows of tracked tables into Flashback Data Archives. When a transaction containing DML on a tracked table commits, this process stores

the pre-image of the changed rows into the Flashback Data Archive. It also keeps metadata on the current rows.

#### force full database caching mode

The caching mode that is manually enabled by executing the `ALTER DATABASE ... FORCE FULL DATABASE CACHING` statement. Unlike in the default caching mode, Oracle Database caches the entire database, LOBs specified with the `NOCACHE` attribute.

#### foreign key

An integrity constraint that requires each value in a column or set of columns to match a value in the unique or primary key for a related table. Integrity constraints for foreign keys define actions dictating database behavior if referenced data is altered.

#### foreign key constraint

A constraint in which Oracle Database enforces the relationship between two tables that contain one or more common columns. The constraint requires that for each value in the column on which the constraint is defined, the value in the other specified other table, and column must match. For example, a referential integrity rule might state that an employee can only work for an existing department.

#### format model

A character literal that describes the format of a datetime in a character string.

#### free list

A linked list called a free list to manage free space in a segment in manual segment space management (MSSM). For a database object that has free space, a free list keeps track of blocks under the high water mark, which is the dividing line between segment space that is used and not yet used. As blocks are used, the database puts blocks on or removes blocks from the free list as needed.

#### full index scan

An index scan in which the database reads only the root and left side branch blocks to find the first leaf block, and then reads the leaf blocks in index sorted order using single block I/O.

#### full outer join

A join between two tables that returns the result of an inner join and the result of a left outer join and a right outer join.

**full table scan**

A scan of table data in which the database sequentially reads all rows from a table and filters out those that do not meet the selection criteria. The database scans all formatted data blocks under the high water mark (HWM).

**function**

A schema object, similar to a PL/SQL procedure, that always returns a single value.

**function-based index**

An index that computes the value of a function or expression involving one or more columns and stores it in the index.

**GDS**

See Global Data Services (GDS)

**GDS catalog**

A metadata repository, located inside an Oracle database, that is associated with a GDS configuration. Every cloud has one and only one catalog.

**GDS configuration**

A set of databases integrated by the GDS framework into a single virtual server that offers one or more global services while ensuring high performance, availability, and optimal utilization of resources.

See also global service.

**GDS pool**

A set of databases within a GDS configuration that provides a unique set of global services and belongs to a specific administrative domain.

**GDS region**

A logical boundary within a GDS configuration that contains database clients and servers that are geographically close to each other.

**Global Data Services (GDS)**

An automated workload management solution for replicated databases. Database services are named representations of one or more database instances. GDS implements the Oracle Database service model across a set of replicated databases.

**global database name**

The combination of the database name (`DB_NAME`) and network domain (`DB_DOMAIN`), for example, `orcl.example.com`. The global database domain is unique within a network.

**global partitioned index**

A B-tree index that is partitioned independently of the partitioning scheme used on the indexed table. A single index partition can point to any or all table partitions.

**global service**

A database service provided by multiple databases synchronized through data replication.

**global service manager**

The central management tool in the Global Data Services framework. At least one global service manager must exist in every GDS region of a GDS configuration.

**global temporary table**

A special temporary table that stores intermediate session-private data for a specific duration.

**granule**

The basic unit of work in parallelism. Oracle Database divides the operation executed in parallel (for example, a table scan, table update, or index creation) into granules. Parallel execution processes execute the operation one granule at a time.

**grid computing**

A computing architecture that coordinates large numbers of servers and storage to act as a single large computer.

**grid infrastructure**

The software that provides the infrastructure for an enterprise grid architecture. In a cluster, this software includes Oracle Clusterware and Oracle ASM. For a standalone server, this software includes Oracle ASM. Oracle Database combines these products into one software installation called the **Grid home**.

**hard parse**

The steps performed by the database to build a new executable version of application code. The database must perform a hard parse instead of a soft parse if the parsed representation of a submitted statement does not exist in the shared pool.

**hash cluster**

A type of table cluster that is similar to an indexed cluster, except the index key is replaced with a hash function. No separate cluster index exists. In a hash cluster, the data is the index.

**hash collision**

Hashing multiple input values to the same output value.

**hash function**

A function that operates on an arbitrary-length input value and returns a fixed-length hash value.

**hash index**

An internal structure in the memoptimize pool that indexes a memory area by primary key.

**hash join**

A join in which the database uses the smaller of two tables or data sources to build a hash table in memory. The database scans the larger table, probing the hash table for the addresses of the matching rows in the smaller table.

**hash key value**

In a hash cluster, an actual or possible value inserted into the cluster key column. For example, if the cluster key is `department_id`, then hash key values could be 10, 20, 30, and so on.

**hash partitioning**

A partitioning strategy that maps rows to partitions based on a hashing algorithm that the database applies to the user-specified partitioning key. The destination of a row is determined by the internal hash function applied to the row by the database. The hashing algorithm is designed to distribute rows evenly across devices so that each partition contains about the same number of rows.

**hash table**

An in-memory data structure that associates join keys with rows in a hash join. For example, in a join of the `employees` and `departments` tables, the join key might be the department ID. A hash function uses the join key to generate a hash value. This hash value is an index in an array, which is the hash table.

**hash value**

In a hash cluster, a unique numeric ID that identifies a bucket. Oracle Database uses a hash function that accepts an infinite number of hash key values as input and sorts them into a finite

number of buckets. Each hash value maps to the database block address for the block that stores the rows corresponding to the hash key value (department 10, 20, 30, and so on).

**hashing**

A mathematical technique in which an infinite set of input values is mapped to a finite set of output values, called hash values. Hashing is useful for rapid lookups of data in a hash table.

**heap-organized table**

A table in which the data rows are stored in no particular order on disk. By default, `CREATE TABLE` creates a heap-organized table.

**hierarchical database**

A database that organizes data in a tree structure. Each parent record has one or more child records, similar to the structure of a file system.

**high water mark (HWM)**

The boundary between used and unused space in a segment.

**hint**

An instruction passed to the optimizer through comments in a SQL statement. The optimizer uses hints to choose an execution plan for the statement.

**hot buffer**

A buffer in the database buffer cache that is frequently accessed and has been recently used.

**human error outage**

An outage that occurs when unintentional or malicious actions are committed that cause data in the database to become logically corrupt or unusable.

**Hybrid Columnar Compression**

A hybrid method that uses row and columnar techniques to compress data in a data block. A logical construct called a compression unit is used to store a set of hybrid columnar-compressed rows.

**hybrid partitioned table**

A table in which some partitions are stored in data file segments and some are stored in external data source.

**IM column store**

An optional SGA area that stores copies of tables and partitions in a columnar format optimized for rapid scans.

**image copy**

A bit-for-bit, on-disk duplicate of a data file, control file, or archived redo log file. You can create image copies of physical files with operating system utilities or RMAN and use either tool to restore them.

**implicit query**

A component of a DML statement that retrieves data without a subquery. An `UPDATE`, `DELETE`, or `MERGE` statement that does not explicitly include a `SELECT` statement uses an implicit query to retrieve the rows to be modified.

**In-Memory Column Store**

See IM column store.

**inactive online redo log file**

An online redo log file that is not required for instance recovery.

**inconsistent backup**

A backup in which some files in the backup contain changes made after the checkpoint. Unlike a consistent backup, an inconsistent backup requires media recovery to be made consistent.

**incremental refresh**

A refresh that processes only the changes to the existing data in a materialized view. This method eliminates the need for a complete refresh.

**incremental-forever backup strategy**

The strategy in which an initial level 0 backup is taken to the Recovery Appliance, with all subsequent incremental backups occurring at level 1. The Recovery Appliance creates a virtual full backup by combining the initial level 0 with subsequent level 1 backups.

**index**

Optional schema object associated with a nonclustered table, table partition, or table cluster. In some cases indexes speed data access.

**index block**

A special type of data block that manages space differently from table blocks.

**index cluster**

An table cluster that uses an index to locate data. The cluster index is a B-tree index on the cluster key.

**index clustering factor**

A measure of the row order in relation to an indexed value such as last name. The more order that exists in row storage for this value, the lower the clustering factor.

**index range scan**

An ordered scan of an index that has the following characteristics:

- One or more leading columns of an index are specified in conditions. A condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.
- 0, 1, or more values are possible for an index key.

**index scan**

The retrieval of a row by traversing an index, using the indexed column values specified by the statement.

**index segment**

A segment that stores data for a nonpartitioned index or index partition.

**index skip scan**

An index scan that uses logical subindexes of a composite index. The database "skips" through a single index as if it were searching separate indexes.

**index unique scan**

An index scan that must have either 0 or 1 rowid associated with an index key. The database performs a unique scan when a predicate references all of the columns in the key of a `UNIQUE` index using an equality operator.

**index-organized table**

A table whose storage organization is a variant of a primary B-tree index. Unlike a heap-organized table, data is stored in primary key order.

**in-doubt distributed transaction**

A distributed transaction in which a two-phase commit is interrupted by any type of system or network failure.

**in-flight transaction**

A transaction that is running when an outage breaks the connection between a client application and the database.

**information system**

A formal system for storing and processing information.

**initialization parameter**

A configuration parameter such as `DB_NAME` or `SGA_TARGET` that affects the operation of a database instance. Settings for initialization parameters are stored in a text-based initialization parameter file or binary server parameter file.

**initialization parameter file**

A text file that contains initialization parameter settings for a database instance.

**inner join**

A join of two or more tables that returns only those rows that satisfy the join condition.

**instance failure**

The termination of a database instance because of a hardware failure, Oracle internal error, or `SHUTDOWN ABORT` statement.

**instance PGA**

The collection of individual PGAs in a database instance.

**instance recovery**

The automatic application of redo log records to uncommitted data blocks when an database instance is restarted after a failure.

**INSTEAD OF trigger**

A trigger that is fired by Oracle Database instead of executing the triggering statement. These triggers are useful for transparently modifying views that cannot be modified directly through DML statements.

**integrity**

See data integrity.

**integrity constraint**

Declarative method of defining a rule for a column. The integrity constraints enforce business rules and prevent the entry of invalid information into tables.

**interested transaction list (ITL)**

Information in a block header that determines whether a transaction was uncommitted when the database began modifying the block. Entries in the ITL describe which transactions have rows locked and which rows in the block contain committed and uncommitted changes.

**interval partition**

An extension of range partitioning that instructs the database to create partitions of the specified range or interval. The database automatically creates the partitions when data inserted into the table exceeds all existing range partitions.

**invisible index**

An index that is maintained by DML operations, but is not used by default by the optimizer. Making an index invisible is an alternative to making it unusable or dropping it.

**invoker's rights PL/SQL procedure**

A procedure that executes in the current user's schema with the current user's privileges.

**Java pool**

An area of memory that stores all session-specific Java code and data within the Java Virtual Machine (JVM).

**Java Publisher (JPublisher)**

A utility that generates Java classes to represent database entities, such as SQL objects and PL/SQL packages, in a Java client program.

**Java stored procedure**

A Java method published to SQL and stored in the database.

**JavaScript object**

An associative array of zero or more pairs of property names and associated JavaScript Object Notation (JSON) values.

**JavaScript Object Notation (JSON)**

A language-independent, text-based data format that can represent objects, arrays, and scalar data.

**job queue process**

An optional background process that runs user jobs, often in batch mode. A job is a user-defined task scheduled to run one or more times.

**join**

A statement that retrieves data from multiple tables specified in the `FROM` clause. Join types include inner joins, outer joins, and Cartesian joins.

**join attribute clustering**

In an attribute-clustered table, clustering that is based on joined columns.

**join condition**

A condition that compares two columns, each from a different table, in a join. The database combines pairs of rows, each containing one row from each table, for which the join condition evaluates to `TRUE`.

**join view**

A view whose definition includes multiple tables or views in the `FROM` clause.

**JSON object**

A JavaScript object literal written as a property listed enclosed in braces.
See also JavaScript Object Notation (JSON).

**JVM**

A virtual processor that runs compiled Java code.

**key**

Column or set of columns included in the definition of certain types of integrity constraints.

**key compression**

See prefix compression.

**key values**

Individual values in a key.

**key-preserved table**

In a join query, a table in which each row appears at most one time in the output of the query.

**large object (LOB)**

See LOB.

**large pool**

Optional area in the SGA that provides large memory allocations for backup and restore operations, I/O server processes, and session memory for the shared server and Oracle XA.

**latch**

A low-level serialization control mechanism used to protect shared data structures in the SGA from simultaneous access.

**latch sleeping**

The phenomenon that occurs when a process releases the CPU before renewing the latch request.

**latch spinning**

The phenomenon that occurs when a process repeatedly requests a latch in a loop.

**leaf block**

In a B-tree index, a lower-level block that stores index entries. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

**left outer join**

The result of a left outer join for table *A* and *B* contains all records of the left table *A*, even if the join condition does not match a record in the right table *B*. For example, if you perform a left outer join of `employees` (left) to `departments` (right), and if some employees are not in a department, then the query returns rows from `employees` with no matches in `departments`.

**library cache**

An area of memory in the shared pool. This cache includes the shared SQL areas, private SQL areas (in a shared server configuration), PL/SQL procedures and packages, and control structures such as locks and library cache handles.

**list partitioning**

A partitioning strategy that uses a list of discrete values as the partition key for each partition. You can use list partitioning to control how individual rows map to specific partitions. By using lists, you can group and organize related sets of data when the key used to identify them is not conveniently ordered.

**listener**

A process that listens for incoming client connection requests and manages network traffic to the database.

**listener registration process (LREG)**

The process that registers information about the database instance and dispatcher processes with the Oracle Net listener.

**literal**

A fixed data value.

**LOB**

Large object. Large Objects include the following SQL data types: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`. These data types are designed for storing data that is large in size.

**local partitioned index**

An index partitioned on the same columns, with the same number of partitions and the same partition bounds as its table. A one-to-one parity exists between index partitions and table partitions.

**local role**

In a CDB, a role that exists only in a single PDB. Unlike a common role, a local role may only contain roles and privileges that apply within the container in which the role exists.

**local temporary tablespace**

A temporary tablespace that resides on local storage and is accessible by a specific database instance. In contrast, a shared shared temporary tablespace resides on shared storage and is accessible by all database instances.

**locale**

Within the context of globalization support, a linguistic and cultural environment in which a system or program is running.

**locally managed tablespace**

A tablespace that uses a bitmap stored in each data file to manage the extents. In contrast, a dictionary-managed tablespace uses the data dictionary to manage space.

**lock**

A database mechanism that prevents destructive interaction between transactions accessing a shared resource such as a table, row, or system object not visible to users. The main categories of locks are DML locks, DDL locks, and latches and internal locks.

**lock conversion**

The automatic conversion of a table lock of lower restrictiveness to one of higher restrictiveness. For example, suppose a transaction issues a `SELECT ... FOR UPDATE` for an employee and later updates the locked row. In this case, the database automatically converts the row share table lock to a row exclusive table lock.

**lock escalation**

A situation that occurs in some databases when numerous locks are held at one level of granularity (for example, rows) and the database raises the locks to a higher level of granularity (for example, table). *Oracle Database never escalates locks.*

**log sequence number**

A number that uniquely identifies a set of redo records in a redo log file. When the database fills one online redo log file and switches to a different one, the database automatically assigns the new file a log sequence number.

**log switch**

The point at which the log writer process (LGWR) stops writing to the active redo log file and switches to the next available redo log file. LGWR switches when either the active redo log file is filled with redo records or a switch is manually initiated.

**log writer process (LGWR)**

The background process responsible for redo log buffer management—writing the redo log buffer to the online redo log. LGWR writes all redo entries that have been copied into the buffer since the last time it wrote.

**logical I/O**

Reads and writes of buffers in the database buffer cache.

**logical read**

A read of a buffer in the database buffer cache.

**logical rowid**

A rowid for an index-organized table. A logical rowid is a base64-encoded representation of a table primary key.

**logical transaction ID**

A globally unique identifier that defines a transaction from the application perspective. The logical transaction ID is bound to the database transaction ID.

**logical volume**

A virtual disk partition.

**logical volume manager (LVM)**

A software package, available with most operating systems, that enables pieces of multiple physical disks to be combined into a single contiguous address space that appears as one disk to higher layers of software.

**lookup table**

A table containing a code column and an associated value column. For example, a job code corresponds to a job name. In contrast to a master table in a pair of master-detail tables, a lookup table is not the means to obtain a detailed result set, such as a list of employees. Rather, a user queries a table such as `employees` for an employee list and then joins the result set to the lookup table.

**lost update**

A data integrity problem in which one writer of data overwrites the changes of a different writer modifying the same data.

**lost write**

A data corruption that occurs when the database thinks that it has written a block to persistent storage, but the block either was not written, or a previous version of the block was written.

**manageability monitor process (MMON)**

The background process that performs many tasks related to the Automatic Workload Repository (AWR). For example, MMON writes when a metric violates its threshold value, taking snapshots, and capturing statistics value for recently modified SQL objects.

**mantissa**

The part of a floating-point number that contains its significant digits.

**manual segment space management (MSSM)**

A legacy space management method that uses a linked list called a free list to manage free space in a segment.

**manual undo management mode**

A mode of the database in which undo blocks are stored in user-managed undo segments. In automatic undo management mode, undo blocks are stored in a system-managed, dedicated undo tablespaces.

**master database**

In replication, the source of the data that is copied to a subscriber database. The replication agent on the master database reads the records from the transaction log for the master database. It forwards changes to replicated elements to the replication agent on the subscriber database. The replication agent on the subscriber database then applies the updates.

**master site**

In a replication environment, a different database with which a materialized view shares data.

**master table**

In a replication environment, the table associated with a materialized view at a master site.

**master-detail tables**

A detail table has a foreign key relationship with a master table. For example, the `employees` detail table has a foreign key to the `departments` master table. Unlike a lookup table, a master table is typically queried and then joined to the detail table. For example, a user may query a department in the `departments` table and then use this result to find the employees in this department.

**materialized view**

A schema object that stores the result of a query. The `FROM` clause of the query can name tables, views, and other materialized views.

See also view.

**media recovery**

The application of redo or incremental backups to a data block or backup data file.

**memoptimize pool**

An SGA pool that stores buffers and related structures for heap-organized tables specified as `MEMOPTIMIZE FOR READ`.

**metadata link**

In a PDB, an internal mechanism that points to a dictionary object definition stored in the root. For example, the `OBJ$` table in each PDB uses a metadata link to point to the definition of `OBJ$` stored in the root.

**metadata-linked common object**

A common object that exists either in the CDB root or an application root. The metadata, rather than the data, is shared by any PDB that contains a metadata link that points to the common object.

**metric**

The rate of change in a cumulative statistic

**mounted database**

An database instance that is started and has the database control file open.

**multitenant architecture**

The architecture that enables an Oracle database to function as a multitenant container database (CDB), which means that it can contain PDBs and application containers.

**multitenant container database (CDB)**

See CDB.

**multithreaded Oracle Database model**

A model that enables Oracle processes to execute as operating system threads in separate address spaces. In threaded mode, some background processes on UNIX and Linux run as processes containing one thread, whereas the remaining Oracle processes run as threads within processes.

**multitier architecture**

An architecture in which one or more application servers provide data for clients and serves as an interface between clients and database servers.

**multiversion consistency model**

A model that enables the database to present a view of data to multiple concurrent users, with each view consistent to a point in time.

**multiversioning**

The ability of the database to simultaneously materialize multiple versions of data.

**mutual exclusion object (mutex)**

A low-level mechanism that prevents an object in memory from aging out or from being corrupted when accessed by concurrent processes.

**natural key**

A meaningful identifier made of existing attributes in a table. For example, a natural key could be a postal code in a lookup table.

**network database**

A type of database, similar to a hierarchical database, in which records have a many-to-many rather than a one-to-many relationship.

**network encryption**

Encrypting data as it travels across the network between a client and server.

**noncircular reuse record**

A control file record that contains critical information that does not change often and cannot be overwritten. Examples of information include tablespaces, data files, online redo log files, and redo threads. Oracle Database never reuses these records unless the corresponding object is dropped from the tablespace.

**nondeferrable constraint**

A constraint whose validity check is never deferred to the end of the transaction. Instead, the database checks the constraint at the end of each statement. If the constraint is violated, then the statement rolls back.

**null**

Absence of a value in a column of a row. Nulls indicate missing, unknown, or inapplicable data.

**object-relational database management system (ORDBMS)**

An RDBMS that implements object-oriented features such as user-defined types, inheritance, and polymorphism.

**object table**

An special kind of table in which each row represents an object.

**object type**

A schema object that abstracts a real-world entity such as a purchase order. Attributes model the structure of the entity, whereas methods implement operations an application can perform on the entity.

**object view**

A virtual object table. Each row in the view is an object, which is an instance of a user-defined data type.

**OLAP**

Online Analytical Processing. OLAP is characterized by dynamic, dimensional analysis of historical data.

**OLAP page pool**

The pool in the UGA that manages OLAP data pages, which are equivalent to data blocks. The page pool is allocated at the start of an OLAP session and released at the end of the session.

**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.

**online redo log**

The set of two or more online redo log files that record all changes made to Oracle Database data files and control file. When a change is made to the database, Oracle Database generates a redo record in the redo buffer. The log writer process (LGWR) process writes the contents of the redo log buffer to the online redo log.

**online redo log group**

An online redo log file and its redundant copies.

**operating system block**

The minimum unit of data that the operating system can read or write.

**operator**

1. In memory management, operators control the flow of data. Examples include sort, hash join, and bitmap merge operators.

2. In SQL, an operator manipulates data items called *operands* or *arguments* and returns a result. Keywords or special characters represent the operators. For example, an asterisk (`*`) represents the multiplication operator.

**optimizer**

Built-in database software that determines the most efficient way to execute a SQL statement by considering factors related to the objects referenced and the conditions specified in the statement.

**optimizer statistics**

Details about the database its object used by the optimizer to select the best execution plan for each SQL statement. Categories include table statistics such as numbers of rows, index statistics such as B-tree levels, system statistics such as CPU and I/O performance, and column statistics such as number of nulls.

**Oracle Application Express**

A Web application development tool for Oracle Database. Oracle Application Express uses built-in features such as user interface themes, navigational controls, form handlers, and flexible reports to accelerate application development.

**Oracle architecture**

Memory and process structures used by Oracle Database to manage a database.

**Oracle ASM**

Oracle Automatic Storage Management (Oracle ASM). A volume manager and a file system for database files. Oracle ASM is Oracle's recommended storage management solution, providing an alternative to conventional volume managers and file systems.

**Oracle ASM allocation unit**

The fundamental unit of allocation within an ASM disk group. An allocation unit is the smallest contiguous disk space that Oracle ASM allocates. One or more allocation units form an Oracle ASM extent.

**Oracle ASM disk**

A storage device that is provisioned to an Oracle ASM disk group. An Oracle ASM disk can be a physical disk or partition, a Logical Unit Number (LUN) from a storage array, a logical volume, or a network-attached file.

**Oracle ASM disk group**

One or more Oracle ASM disks managed as a logical unit. I/O to a disk group is automatically spread across all the disks in the group.

**Oracle ASM extent**

A section of an Oracle ASM file. An Oracle ASM file consists of one or more file extents. Each Oracle ASM extent consists of one or more allocation units on a specific disk.

**Oracle ASM file**

A file stored in an Oracle ASM disk group. The database can store data files, control files, online redo log files, and other types of files as Oracle ASM files.

**Oracle ASM instance**

A special Oracle instance that manages Oracle ASM disks. Both the Oracle ASM instance and the database instances require shared access to the disks in an Oracle ASM disk group. Oracle ASM instances manage the metadata of the disk group and provide file layout information to the database instances.

**Oracle Automatic Storage Management (Oracle ASM)**

See Oracle ASM.

**Oracle base**

The Oracle base directory is the database home directory for Oracle Database installation owners. The directory, which is specified by the `ORACLE_BASE` environment variable, is the root of the Oracle directory tree.

**Oracle base configuration directory**

A directory that contains the instance-specific configuration files.
For a read-write Oracle home, the Oracle base home (`ORACLE_BASE_HOME`), Oracle base configuration directory (`ORACLE_BASE_CONFIG`), and Oracle home are the same. For a read-only Oracle home, the Oracle base configuration directory contains configuration files shared by all Oracle homes in an Oracle base home. To prevent naming collisions, each file name contains the SID (system identifier).

**Oracle base home**

A directory that stores configuration files for the database instances associated with an Oracle home.
For a read-write Oracle home, the Oracle base home (`ORACLE_BASE_HOME`), Oracle base configuration directory (`ORACLE_BASE_CONFIG`), and Oracle home are the same. For a read-only Oracle home, the Oracle base home is a home-specific directory located in the `homes/home_name` subdirectory of the Oracle base.

**Oracle Clusterware**

A set of components that enables servers to operate together as if they were one server. Oracle Clusterware is a requirement for using Oracle RAC and it is the only clusterware that you need for platforms on which Oracle RAC operates.

**Oracle Connection Manager**

A router through which a client connection request may be sent either to its next hop or directly to the database server.

**Oracle Data Redaction**

A feature of Oracle Advanced Security that enables you to mask (redact) data that is queried by low-privileged users or applications.

**Oracle database**

A set of files, located on disk, that store data. Because a database instance and a database are so closely connected, the term *Oracle database* is often used to refer to both instance and database.

**Oracle Database Vault**

A database security feature that controls when, where, and how databases, data, and applications are accessed.

**Oracle Developer Tools for Visual Studio .NET**

A set of application tools integrated with the Visual Studio .NET environment. These tools provide GUI access to Oracle functionality, enable the user to perform a wide range of application development tasks, and improve development productivity and ease of use.

**Oracle Enterprise Manager**

A system management tool that provides centralized management of an Oracle database environment.

**Oracle Flashback Technology**

A group of features that supports viewing past states of data, and winding data back and forth in time, without needing to restore backups.

**Oracle Flex Clusters**

A large cluster configured using Oracle Clusterware and Oracle Real Application Clusters. These clusters contain two types of nodes arranged in a hub-and-spoke architecture: Hub Nodes and Leaf Nodes.

**Oracle Globalization Development Kit (GDK)**

A development toolkit that includes comprehensive programming APIs for both Java and PL/SQL, code samples, and documentation that address many of the design, development, and deployment issues encountered while creating global applications.

**Oracle home**

The operating system location of an Oracle Database installation. An Oracle home can be read/write or read-only.

**Oracle JDeveloper**

An integrated development environment (IDE) for building service-oriented applications using the latest industry standards for Java, XML, Web services, and SQL.

**Oracle JVM**

A standard, Java-compatible environment that runs any pure Java application.

**Oracle Managed Files**

A database file naming strategy that enables database administrators to specify operations in terms of database objects rather than file names. Oracle Managed Files eliminates the need for administrators to directly manage the operating system files in a database.

**Oracle Multimedia**

A technology that enables Oracle Database to store, manage, and retrieve images, DICOM format medical images and other objects, audio, video, or other heterogeneous media data in an integrated fashion with other enterprise information.

**Oracle Multitenant**

A database option that enables you to create multiple PDBs in a CDB.

### Oracle Net

Communication software that enables a network session between a client application and an Oracle database. After a network session is established, Oracle Net acts as a data courier for the client application and the database.

### Oracle Net Listener

A process that resides on the server whose responsibility is to listen for incoming client connection requests and manage the traffic to the server. When a client requests a network session with a database, Oracle Net Listener (typically called *the listener*) receives the request. If the client information matches the listener information, then the listener grants a connection to the database server.

### Oracle Net Services

A suite of networking components that provide enterprise-wide connectivity solutions in distributed, heterogeneous computing environments. Oracle Net Services includes Oracle Net, listener, Oracle Connection Manager, Oracle Net Configuration Assistant, and Oracle Net Manager.

### Oracle Optimal Flexible Architecture (OFA)

A set of configuration guidelines created to ensure well-organized Oracle installations.

### Oracle process

A unit of execution that runs the Oracle database code. The process execution architecture depends on the operating system. Oracle processes include server processes and background processes.

### Oracle RAC

Oracle Real Application Clusters. Option that allows multiple concurrent database instances to share a single physical database.

### Oracle Real Application Clusters

See Oracle RAC.

### Oracle Sharding

A feature for OLTP applications that enables distribution and replication of data across a pool of Oracle databases in a shared-nothing architecture. Applications access the pool as a single, logical database called a sharded database (SDB).

### Oracle Spatial and Graph

A set of advanced features for spatial data and analysis and for physical, logical, network, and social and semantic graph applications. The spatial features provide a schema and functions

that facilitate the storage, retrieval, update, and query of collections of spatial features in an Oracle database.

**Oracle SQL**

An implementation of the ANSI standard for SQL. Oracle SQL supports numerous features that extend beyond standard SQL.

**Oracle Text (Text)**

A full-text retrieval technology integrated with Oracle Database.

**Oracle Virtual Private Database (VPD)**

A security feature that enables you to create security policies to control 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 a VPD security policy was applied.

**Oracle XA**

An external interface that allows global transactions to be coordinated by a transaction manager other than Oracle Database.

**Oracle XML DB**

A set of Oracle Database technologies related to high-performance XML manipulation, storage, and retrieval. Oracle XML DB provides native XML support by encompassing both SQL and XML data models in an interoperable manner.

**Oracle XML Developer's Kit (XDK)**

A developer toolkit that contains the basic building blocks for reading, manipulating, transforming, and viewing XML documents, whether on a file system or in a database. APIs and tools are available for Java, C, and C++. The production Oracle XDK comes with a commercial redistribution license.

**outer join**

A join that returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

**parallel execution**

The application of multiple CPU and I/O resources to the execution of a single database operation.

**parse lock**

A lock is held by a SQL statement or PL/SQL program unit for each schema object that it references. Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped.

**partial index**

An index that is correlated with the indexing properties of an associated partitioned table.

**partition**

A piece of a table or index that shares the same logical attributes as the other partitions. For example, all partitions in a table share the same column and constraint definitions. Each partition is an independent object with its own name and optionally its own storage characteristics.

**partition elimination**

The exclusion of partitions from a query plan. Whether the optimizer can eliminate partitions from consideration depends on the query predicate. A query that uses a local prefixed index always allows for index partition elimination, whereas a query that uses a local nonprefixed index might not.

**partition key**

A set of one or more columns that determines the partition in which each row in a partitioned table should go. Each row is unambiguously assigned to a single partition.

**partitioned index**

An index that is divided into smaller and more manageable pieces. Like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability.

**partitioned table**

A table that has one or more partitions, each of which is managed individually and can operate independently of the other partitions.

**partitioning**

The ability to decompose very large tables and indexes into smaller and more manageable pieces called partitions.

**PDB**

In a multitenant container database (CDB), a portable collection of schemas, schema objects, and nonschema objects that appears to an Oracle Net client as a separate database.

**PDB administrator**

A database administrator who manages one or more PDBs. A CDB administrator manages the whole CDB.

**PDB restore point**

Within a CDB, a restore point that usable only for a specific PDB. In contrast, a CDB restore point is usable by all PDBs.

**performance profile**

A specified share of system resources, CPU, parallel execution servers, and memory for a PDB or set of PDBs.

**permanent tablespace**

A tablespace that contains persistent schema objects. Every tablespace that is not a temporary tablespace is a permanent tablespace.

**PGA**

Program global area. A memory buffer that contains data and control information for a server process.

See also SGA.

**physical guess**

The physical rowid of an index entry when it was first made. Oracle Database can use physical guesses to probe directly into the leaf block of any index-organized table, bypassing the primary key search.

**PL/SQL**

Procedural Language/SQL. The Oracle Database procedural language extension to SQL. PL/SQL enables you to mix SQL statements with programmatic constructs such as procedures, functions, and packages.

**PL/SQL anonymous block**

A PL/SQL block that appears in an application, but is not named or stored in the database. In many applications, PL/SQL blocks may appear wherever SQL statements can appear.

**PL/SQL collection**

An ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection.

**PL/SQL engine**

The tool used to define, compile, and run PL/SQL program units. This engine is a special component of many Oracle products, including Oracle Database.

**PL/SQL function**

A schema object that consists of a set of SQL statements and other PL/SQL constructs, grouped together, stored in the database, and run as a unit to solve a specific problem or perform a set of related tasks, and that always returns a single value to the caller.

**PL/SQL function result cache**

A subset of the server result cache that stores function result sets.

**PL/SQL package**

A logical grouping of related PL/SQL types, variables, and subprograms.

**PL/SQL procedure**

A schema object that consists of a set of SQL statements and other PL/SQL constructs, grouped together, stored in the database, and run as a unit to solve a specific problem or perform a set of related tasks.

**PL/SQL record**

A composite variable that can store data values of different types, similar to a `struct` type in C, C++, or Java. Records are useful for holding data from table rows, or specific columns from table rows.

**PL/SQL subprogram**

A named PL/SQL block that can be invoked with a set of parameters

**plan generator**

The part of the optimizer that tries different access paths, join methods, and join orders for a given query block to find the plan with the lowest cost.

**pluggable database (PDB)**

See PDB.

**population**

The transfer of data into the IM column store. Population does not insert *new* data into the database; rather, it brings *existing* data into memory and stores it in columnar format.

**pragma**

A directive that instructs the compiler to perform a compilation option. For example, the pragma `AUTONOMOUS_TRANSACTION` instructs the database that this procedure, when executed, is to be executed as a new autonomous transaction that is independent of its parent transaction.

**precision**

The total number of digits in a floating-point number. You specify a fixed-point number in the form `NUMBER(p,s)`, where *p* represents the precision.

**precompiler**

A programming tool that enables you to embed SQL statements in a high-level source program written in a language such as C, C++, or COBOL.

**predicate**

The `WHERE` condition in a SQL statement.

**prefix compression**

The elimination of repeated occurrence of primary key column values in an index-organized table. Prefix compression was formerly known as *key compression*.

**primary key**

The column or set of columns that uniquely identifies a row in a table. Only one primary key can be defined for each table.

**primary key constraint**

An integrity constraint that disallows duplicate values and nulls in a column or set of columns.

**private SQL area**

An area in memory that holds a parsed statement and other information for processing. The private SQL area contains data such as bind variable values, query execution state information, and query execution work areas.

**private temporary table**

A memory-only temporary table whose data and metadata is session-private.

**privilege**

The right to run a particular type of SQL statement, or the right to access an object that belongs to another user, run a PL/SQL package, and so on. The types of privileges are defined by Oracle Database.

**privilege analysis**

A security mechanism that captures privilege usage for a database according to a specified condition. For example, you can find the privileges that a user exercised during a specific database session.

**procedural language**

A language that describes *how* things should be done, not *what* should be done (as in declarative languages). C++ and Java are examples of procedural languages.

**process**

A mechanism in an operating system that can run a series of steps. By dividing the work of Oracle Database and database applications into several processes, multiple users and applications can connect to a single database instance simultaneously.

See also background process; Oracle process; client process.

**process monitor (PMON)**

The background process that detects the termination of other background processes. If a server or dispatcher process terminates abnormally, then the process monitor (PMON) group is responsible for performing process recovery.

**process monitor (PMON) group**

The group of background processes that is responsible for the monitoring and cleanup of other processes. The PMON group includes process monitor (PMON), Cleanup Main Process (CLMN), and Cleanup Helper Processes (CL*nn*).

**program global area (PGA)**

See PGA.

**program interface**

The software layer between a database application and Oracle Database.

**protected database**

A client database whose backups are managed by a Recovery Appliance.

**protection policy**

A group of attributes that control how a Recovery Appliance stores and maintains backup data. Each protected database is assigned to exactly one protection policy, which controls all aspects of backup processing for that client.

**proxy PDB**

A PDB that references a PDB in a remote CDB using a database link. The remote PDB is called a referenced PDB.

**pseudocolumn**

A column that is not stored in a table, yet behaves like a table column.

**query**

An operation that retrieves data from tables or views. For example, `SELECT * FROM employees` is a query.

See also implicit query; subquery.

**query block**

A top-level `SELECT` statement, subquery, or unmerged view.

**query coordinator**

In parallel execution, the user session or shadow process that coordinates the parallel execution servers. The parallel execution servers performs each operation in parallel if possible. When the parallel servers are finished executing the statement, the query coordinator performs any portion of the work that cannot be executed in parallel. Finally, the query coordinator returns any results to the user.

**query optimization**

The process of choosing the most efficient means of executing a SQL statement.

**query plan**

The execution plan used to execute a query.

**query rewrite**

An optimization technique that transforms a user request written in terms of master tables into a semantically equivalent request that includes materialized views.

**query transformer**

An optimizer component that decides whether it can rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost.

**R**

A language and environment for statistical computing and graphics.

**range partitioning**

A type of partitioning in which the database maps rows to partitions based on ranges of values of the partitioning key. Range partitioning is the most common type of partitioning and is often used with dates.

**RDMA**

Remote Direct Memory Access. A communication protocol that enables a client to transfer data from the memory (DRAM) of the database server over the network without consuming server-side CPU.

**read committed isolation level**

An isolation level that guarantees that a query executed by a transaction sees only data committed before the query—not the transaction—began.

**read consistency**

A consistent view of data seen by a user. For example, in statement-level read consistency the set of data seen by a SQL statement remains constant throughout statement execution.

See also data concurrency; data consistency.

**read-only database**

A database that is available for queries only and cannot be modified.

**read-only database instance**

A database instance that cannot process DML and does not support client connections.

**read-only isolation level**

An isolation level that is similar to the serializable isolation level, with one exception: read-only transactions do not permit data to be modified in the transaction unless the user is `SYS`.

**read/write database instance**

A database instance that can process DML and supports direct client connections. By default, a database instance is read/write.

**real-time redo transport**

The continuous transfer of redo changes from the SGA of a protected database to a Recovery Appliance. Real-time redo transport enables RMAN to provide a recovery point objective near 0. Typically, RMAN can recover to within a second of the time when the failure occurred. Protected databases write redo entries directly from the SGA to the Recovery Appliance as they are generated.

**recoverable error**

A class of errors that arise because of an external system failure, independently of the application session logic that is executing. Recoverable errors occur following planned and unplanned outages of networks, nodes, storage, and databases. An example of a nonrecoverable error is submission of invalid data values.

**recoverer process (RECO)**

In a distributed database, the background process that automatically resolves failures in distributed transactions.

**Recovery Appliance**

Shortened name for Zero Data Loss Recovery Appliance. Recovery Appliance is an Oracle Engineered System specifically designed to protect Oracle databases. Integrated with RMAN, it enables a centralized, incremental-forever backup strategy for hundreds to thousands of databases across the enterprise, using cloud-scale, fully fault-tolerant hardware and storage.

**Recovery Appliance Backup Module**

An Oracle-supplied SBT library that RMAN uses to send backups of protected databases over the network to the Recovery Appliance. The library must be installed in each Oracle home used by a protected database.

The module functions as an SBT media management library that RMAN references when allocating or configuring a channel for backup to the Recovery Appliance. RMAN performs all backups to the Recovery Appliance, and all restores of complete backup sets, using this module.

**Recovery Appliance metadata database**

The Oracle database that runs inside of the Recovery Appliance. This database stores configuration data such as user definitions, protection policy definitions, and client database

definitions. The metadata database also stores backup metadata, including the contents of the delta store.

**Recovery Appliance storage location**

A set of Oracle ASM disk groups within Recovery Appliance that stores backups. A storage location can be shared among multiple protected databases. Every Recovery Appliance contains the default Recovery Appliance storage location named `DELTA`.

**recovery catalog**

A centralized backup repository located in an Oracle database. The recovery catalog contains metadata about RMAN backups.

**Recovery Manager (RMAN)**

See RMAN.

**recovery window goal**

The time interval within which a protected database must be recoverable to satisfy business requirements. For each protected database in a protection policy, the Recovery Appliance attempts to ensure that the oldest backup on disk is able to support a point-in-time recovery to any time within the specified interval (for example, the past 7 days), counting backward from the current time.

**recursive SQL**

SQL that the database executes in the background to obtain space for database objects. You can think of recursive SQL as "side effect" SQL.

**redo log**

A set of files that protect altered database data in memory that has not been written to the data files. The redo log can consist of two parts: the online redo log and the archived redo log.

**redo log buffer**

Memory structure in the SGA that stores redo entries—a log of changes made to the database. The database writes the redo entries stored in the redo log buffers to an online redo log file, which the database uses when instance recovery is necessary.

**redo record**

A record in the online redo log that holds a group of change vectors, each of which describes a change made to a data block. Each redo log file consists of redo records.

**redo thread**

The redo generated by a database instance.

**reference partitioning**

A partitioning strategy in which a child table is solely defined through the foreign key relationship with a parent table. For every partition in the parent table, exactly one corresponding partition exists in the child table.

**referenced key**

In a foreign key relationship, the primary or unique key to which the foreign key refers. For example, in the common schema, the `employees.department_id` column is a foreign key, and the `departments.department_id` column is the referenced key.

**referenced object**

In a schema object dependency, the object that is referenced by another object's definition. For example, if the definition of object A references object B, then B is a referenced object for A.

**referenced PDB**

The PDB that is referenced by a proxy PDB. A local PDB is in the same CDB as its referenced PDB, whereas a remote PDB is in a different CDB.

**referential integrity**

A rule defined on a key in one table that guarantees that the values in that key match the values in a key in a related table (the referenced value).

**refreshable clone PDB**

A read-only clone that can periodically synchronize with its source PDB. Depending on the value in the `REFRESH MODE` clause, the synchronization occurs either automatically or manually.

**relation**

A set of tuples.

**relational database**

A database that conforms to the relational model, storing data in a set of simple relations.

**relational database management system (RDBMS)**

A management system that moves data into a relational database, stores the data, and retrieves it so that applications can manipulate it.

**replay context**

In Application Continuity, opaque information that the database returns to the client driver during normal application run time.

**replication**

The process of sharing database objects and data at multiple databases.

**reserved pool**

A memory area in the shared pool that Oracle Database can use to allocate large contiguous chunks of memory.

**resource plan**

A container for resource plan directives that specify how resources are allocated to resource consumer groups.

**resource plan directive**

A set of limits and controls for CPU, physical I/O, or logical I/O consumption for sessions in a consumer group.

**restore point**

A user-defined a name associated with an SCN of the database corresponding to the time of the creation of the restore point.

**result set**

The set of data retrieved from execution of a `SELECT` statement.

**reverse key index**

A type of B-tree index that physically reverses the bytes of each index key while keeping the column order. For example, if the index key is `20`, and if the two bytes stored for this key in hexadecimal are `C1,15` in a standard B-tree index, then a reverse key index stores the bytes as `15,C1`.

**right outer join**

The result of a right outer join for table *A* and *B* contains all records of the right table *B*, even if the join condition does not match a record in the left table *A*. For example, if you perform a right outer join of `employees` (left) to `departments` (right), and if some departments contain no employees, then the query returns rows from `departments` with no matches in `employees`.

**RMAN**

Recovery Manager. An Oracle Database utility that backs up, restores, and recovers Oracle databases.

**role**

A set of privileges that can be granted to database users or to other roles.

**row**

A set of column information corresponding to a single record in a table. The database stores rows in data blocks.

**row chaining**

A situation in which Oracle Database must store a row in a series or chain of blocks because it is too large to fit into a single block.

**row lock**

A lock on a single row of table. A transaction acquires a row lock for each row modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or `SELECT ... FOR UPDATE` statement.

**row major format**

A type of table storage in which all columns of one row are stored together, followed by all columns of the next row, and so on.

**row migration**

A situation in which Oracle Database moves a row from one data block to another data block because the row grows too large to fit in the original block.

**row piece**

A row is stored in a variable-length record. This record is divided into one or more row pieces. Each row piece has a row header and column data.

**row set**

A set of rows returned by a step in an execution plan.

**row source**

An iterative control structure that processes a set of rows and produces a row set.

**row source generator**

Software that receives the optimal plan from the optimizer and outputs the execution plan for the SQL statement.

**row trigger**

A trigger that fires each time the table is affected by the triggering statement. For example, if a statement updates multiple rows, then a row trigger fires once for each row affected by the `UPDATE`.

**rowid**

A globally unique address for a row in a database.

**sample schemas**

A set of interlinked schemas that enable Oracle documentation and Oracle instructional materials to illustrate common database tasks.

**savepoint**

A named SCN in a transaction to which the transaction can be rolled back.

**scale**

In a floating-point number, the number of digits from the decimal point to the least significant digit. You specify a fixed-point number in the form `NUMBER(p,s)`, where *s* represents the scale.

**schema**

A named collection of database objects, including logical structures such as tables and indexes. A schema has the name of the database user who owns it.

**schema object**

A logical structure of data stored in a schema. Examples of schema objects are tables, indexes, sequences, and database links.

**schema object dependency**

The referencing of one object by another object. For example, a view contains a query that references tables or views, or a PL/SQL subprogram invokes other subprograms.

**SCN**

System Change Number. A database ordering primitive. The value of an SCN is the logical point in time at which changes are made to a database.

**secondary index**

An index on an index-organized table. In a sense, it is an index on an index.

**SecureFiles LOB storage**

SecureFiles LOB storage is the default storage mechanism for LOBs. The `SECUREFILE` LOB parameter enables advanced features, including compression and deduplication (part of the Advanced Compression Option) and encryption (part of the Advanced Security Option).

**security policy**

A set of methods for protecting a database from accidental or malicious destruction of data or damage to the database infrastructure.

**seed PDB**

In a multitenant container database (CDB), a default pluggable database (PDB) that the system uses as a template for user-created PDBs. A PDB seed is either the system-supplied `PDB$SEED` or an application seed.

**segment**

A set of extents allocated for a specific database object such as a table, index, or table cluster. User segments, undo segments, and temporary segments are all types of segments.

**select list**

In a `SELECT` statement, the list of expressions that appears after the `SELECT` keyword and before the `FROM` clause.

**selectivity**

A value indicating the proportion of a row set retrieved by a predicate or combination of predicates, for example, `WHERE last_name = 'Smith'`. A selectivity of `0` means that no rows pass the predicate test, whereas a value of `1` means that all rows pass the test.

The adjective *selective* means roughly "choosy." Thus, a highly selective query returns a low proportion of rows (selectivity close to `0`), whereas an unselective query returns a high proportion of rows (selectivity close to `1`).

**self join**

A join of a table to itself.

**self-referential integrity constraint**

A constraint in which a foreign key references a parent key in the same table. For example, a constraint could ensure that every value in the `employees.manager_id` column corresponds to an existing value in the `employees.employee_id` column.

**sequence**

A schema object that generates a serial list of unique numbers for table columns.

**serial execution**

A single server process performs all necessary processing for the sequential execution of a SQL statement.

**serializability**

A transaction isolation model that enables a transaction to operate in an environment that makes it appear as if no other users were modifying data in the database.

**serializable isolation level**

A level of isolation that guarantees that a transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself.

**server**

In a client/server architecture, the computer that runs Oracle software and handles the functions required for concurrent, shared data access. The server receives and processes the SQL and PL/SQL statements that originate from client applications.

**server parameter file**

A server-side binary file containing initialization parameter settings that is read and written to by the database.

**server process**

An Oracle process that communicates with a client process and Oracle Database to fulfill user requests. The server processes are associated with a database instance, but are not part of the instance.

**server result cache**

A memory pool within the shared pool. This memory pool consists of the SQL query result cache—which stores results of SQL queries—and the PL/SQL function result cache, which stores values returned by PL/SQL functions.

**service handler**

In Oracle Net, a dedicated server process or dispatcher that acts as a connection point to a database.

**service name**

In Oracle Net, the logical representation of a service used for client connections.

**service registration**

In Oracle Net, a feature by which the listener registration process (LREG) dynamically registers instance information with a listener, which enables the listener to forward client connection requests to the appropriate service handler.

**service-oriented architecture (SOA)**

A multitier architecture relying on services that support computer-to-computer interaction over a network.

**session**

A logical entity in the database instance memory that represents the state of a current user login to a database. A single connection can have 0, 1, or more sessions established on it.

**SGA**

System global area. A group of shared memory structures that contain data and control information for one Oracle database instance.

**shadow extent**

An extent in a shadow tablespace. A one-to-one mapping exists between a shadow extent and a tracked data file.

**shadow lost write protection**

A data protection feature that uses a shadow tablespace to detect a lost write automatically in any tracked data file.

**shadow tablespace**

A tablespace enabled for shadow lost write protection. The tablespace tracks the SCNs and other metadata for data blocks stored in a tracked data file. A single shadow tablespace may contain descriptive data for many regular data files.

**shard**

A single database participating in a sharding configuration.

**shard catalog database**

A database that stores the sharded database (SDB) configuration data and provides other functionality, such as cross shard queries and centralized management.

**shard director**

A GDS infrastructure component that uses the global service manager to provide direct routing of requests from the application tier to an individual shard.

**sharded database (SDB)**

In a sharding architecture, collection of shards that appear to applications as a single logical database.

**sharded table**

A table that is split horizontally across a sharded database (SDB), so that each shard contains the table with the same columns but a different subset of rows.

**sharding**

A data tier architecture in which data is horizontally partitioned across independent databases. Sharding is a shared-nothing database architecture because shards do not share physical resources such as CPU, memory, or storage devices. Shards are also loosely coupled in terms of software; they do not run clusterware.

**sharding key**

A partitioning key for a sharded table.

**share lock**

A lock that permits the associated resource to be shared by multiple transactions, depending on the operations involved. Multiple transactions can acquire share locks on the same resource.

**shared pool**

Portion of the SGA that contains shared memory constructs such as shared SQL areas.

**shared server**

A database configuration that enables multiple client processes to share a small number of server processes.

See also dedicated server.

**shared SQL area**

An area in the shared pool that contains the parse tree and execution plan for a SQL statement. Only one shared SQL area exists for a unique statement.

**shared temporary tablespace**

A temporary tablespace that resides on shared storage and is accessible by all database instances. Starting in Oracle Database 12*c* Release 2 (12.2), temporary tablespaces are either shared or local. In previous releases, all temporary tablespaces were shared temporary tablespaces.

**shared undo mode**

The use of a single set of undo data files for an entire CDB.

**simple database operation**

A single SQL statement, or a single PL/SQL procedure or function.

**simple trigger**

A trigger on a table that enables you to specify actions for exactly one timing point. For example, the trigger might fire before the firing statement.

**single-level partitioning**

A partitioning strategy that uses only one method of data distribution, for example, only list partitioning or only range partitioning.

**site failure**

An event that causes all or a significant portion of an application to stop processing or slow to an unusable service level.

**smallfile tablespace**

A tablespace that can contain multiple data files or temp files, but the files cannot be as large as in a bigfile tablespace.

**soft parse**

The reuse of existing code when the parsed representation of a submitted SQL statement exists in the shared pool and can be shared.

See also hard parse.

**software code area**

A portion of memory that stores code that is being run or can be run.

**sorted hash cluster**

A hash cluster that stores the rows corresponding to each value of the hash function in such a way that the database can efficiently return them in sorted order. The database performs the optimized sort internally.

**SQL**

Structured Query Language. A nonprocedural language to access a relational database. Users describe in SQL what they want done, and the SQL language compiler automatically generates a procedure to navigate the database and perform the task. **Oracle SQL** includes many extensions to the ANSI/ISO standard SQL language.

See also SQL*Plus; PL/SQL.

**SQL Developer**

A graphical version of SQL*Plus, written in Java, that supports development in SQL and PL/SQL.

**SQL parsing**

This stage of SQL processing that involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.

**SQL plan baseline**

In SQL plan management, a set of one or more accepted plans for a repeatable SQL statement. The effect of a SQL plan baseline is that the optimizer limits its choice to a verified plan in the baseline.

**SQL plan management**

A preventative mechanism that enables the optimizer to automatically manage execution plans, ensuring that the database uses only verified plans.

**SQL profile**

A set of auxiliary information built during automatic tuning of a SQL statement. A SQL profile is to a SQL statement what statistics are to a table. The optimizer can use SQL profiles to

improve cardinality and selectivity estimates, which in turn leads the optimizer to select better plans.

**SQL query result cache**

A subset of the server result cache that stores the results of queries and query fragments.

**SQL*Plus**

Oracle tool used to run SQL statements against Oracle Database.

**SQLJ**

An ANSI standard for embedding SQL statements in Java programs. You can combine SQLJ programs with JDBC.

**standby database**

An independent copy of a production database that you can use for disaster protection in a high availability environment.

**star schema**

A relational schema whose design represents a dimensional data model. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys.

See also dimension table; fact table.

**state object**

A session-level structure that contains metadata about the status of database resources such as processes, sessions, and transactions in the SGA.

**statement trigger**

A trigger that is fired once on behalf of the triggering statement, regardless of the number of rows affected by the triggering statement.

**statement-level atomicity**

The characteristic of a SQL statement as an atomic unit of work that either completely succeeds or completely fails.

**statement-level read consistency**

The guarantee that data returned by a single query is committed and consistent for a single point in time.

**statement-level rollback**

A database operation in which the effects of an unsuccessful SQL statement are rolled back because the statement caused an error during execution.

**stored procedure**

A named PL/SQL block or Java program that Oracle Database stores in the database. Applications can call stored procedures by name.

**Streams pool**

A memory pool that stores buffered queue messages.

**Structured Query Language (SQL)**

See SQL.

**subquery**

A query nested within another SQL statement. Unlike implicit queries, subqueries use a `SELECT` statement to retrieve data.

**summary**

In a data warehouse, an aggregate view that reduces query time by precalculating joins and aggregation operations and storing the results in a table.

**surrogate key**

A system-generated incrementing identifier that ensures uniqueness within a table. Typically, a sequence generates surrogate keys.

**synonym**

An alias for a schema object. You can use synonyms to provide data independence and location transparency.

**system change number (SCN)**

See SCN.

**system container**

The container that includes the CDB root and all PDBs in the CDB.

**system event trigger**

An event trigger caused by events such as error messages, or database instance startup and shutdown.

**system global area (SGA)**

See SGA.

**system monitor process (SMON)**

The background process in charge of a variety of system-level cleanup duties, including instance recovery, recovering terminated transactions that were skipped during instance recovery, Cleaning up unused temporary segments, and Coalescing contiguous free extents within dictionary-managed tablespaces.

**table**

Basic unit of data storage in Oracle Database. Data in tables is stored in rows and columns.

**table cluster**

A schema object that contains data from one or more tables, all of which have one or more columns in common. In table clusters, the database stores together all the rows from all tables that share the same cluster key.

**table compression**

The compression of data segments to reduce disk space in a heap-organized table or table partition.

**table function**

A user-defined PL/SQL function that returns a collection of rows (a nested table or varray). You can select from this collection as if it were a database table by invoking the table function inside the `TABLE` clause in a `SELECT` statement.

**table lock**

A lock on a table that is acquired by a transaction when a table is modified by an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `SELECT ... FOR UPDATE`, or `LOCK TABLE` statement.

**tablespace**

A database storage unit that groups related logical structures together. The database data files are stored in tablespaces.

**tablespace set**

In Oracle Sharding, tablespaces that are distributed across a sharded database (SDB) and managed as a unit.

**temp file**

A file that belongs to a temporary tablespace. The temp files in temporary tablespaces cannot contain permanent database objects.

**temporary segment**

A segment created by Oracle Database when a SQL statement needs a temporary database area to complete execution.

**temporary table**

A table that holds an intermediate result set for the duration of a transaction or a session. Only the current session can see the data in temporary tables.

**temporary tablespace**

A tablespace that can only contain transient data that persists only for the duration of a session. No permanent schema objects can reside in a temporary tablespace.

Every temporary tablespace is either a shared temporary tablespace or a local temporary tablespace. Unless otherwise stated, the term *temporary tablespace* means *shared temporary tablespace*.

**temporary undo segment**

An optional space management container for temporary undo data only.

**trace file**

An administrative file that contain diagnostic data used to investigate problems. Oracle Database writes trace files to ADR.

**tracked data file**

A data file whose block SCNs are tracked by a shadow tablespace, enabling shadow lost write protection.

**transaction**

Logical unit of work that contains one or more SQL statements. All statements in a transaction commit or roll back together. The use of transactions is one of the most important ways that a database management system differs from a file system.

**transaction entry**

Space in the block header that is required for every transaction that updates the block. In data blocks allocated to segments that support transactional changes, free space can also hold transaction entries when the header space is depleted.

**Transaction Guard**

A database feature that uses a logical transaction ID to prevent the possibility of a client application submitting duplicate transactions after a recoverable error.

**transaction ID**

An identifier is unique to a transaction and represents the undo segment number, slot, and sequence number.

**transaction idempotence**

The ability to return a guaranteed outcome for a transaction: whether it committed and whether the call was completed.

**transaction-level read consistency**

The guarantee of read consistency to all queries in a transaction. Each statement in a transaction sees data from the same point in time, which is the time at which the transaction began.

**transaction name**

An optional, user-specified tag that serves as a reminder of the work that the transaction is performing. Name a transaction with the `SET TRANSACTION ... NAME` statement.

**transaction recovery**

A phase of instance recovery in which uncommitted transactions are rolled back.

**transaction table**

The data structure within an undo segment that holds the transaction identifiers of the transactions using the undo segment.

**transition point**

The high value of the range partitions determined by the range partition key value.

**Transparent Data Encryption**

A database feature that encrypts individual table columns or a tablespace. When a user inserts data into an encrypted column, the database automatically encrypts the data. When users select the column, the data is decrypted. This form of encryption is transparent, provides high performance, and is easy to implement.

**transportable tablespace**

A tablespace that you can copy or move between databases. Oracle Data Pump provides the infrastructure for transportable tablespaces.

**trigger**

A PL/SQL or Java procedure that fires when a table or view is modified or when specific user or database actions occur. Procedures are explicitly run, whereas triggers are implicitly run.

**tuple**

An unordered set of attribute values.

**two-phase commit mechanism**

A mechanism in a distributed database that guarantees that all databases participating in a distributed transaction either all commit or all undo the statements in the transaction.

**UGA**

User global area. Session memory that stores session variables, such as logon information, and can also contain the OLAP pool.

**undo data**

Records of the actions of transactions, primarily before they are committed. The database can use undo data to logically reverse the effect of SQL statements. Undo data is stored in undo segments.

**undo retention period**

The minimum amount of time that the database attempts to retain old undo data before overwriting it.

**undo segment**

A segment in an undo tablespace.

**undo tablespace**

A tablespace containing undo segments when automatic undo management mode is enabled.

**Unicode**

A universal encoded character set that can store information in any language using a single character set.

**unified audit policy**

A policy that you can use to configure auditing on SQL statements, system privileges, schema objects, roles, administrative and non-administrative users, application context values, and policy creations for various applications and events.

**unified audit trail**

An audit trail provides unified storage for audit records from all types of auditing.

**unique key**

A single column with a unique key constraint.

**unique key constraint**

An integrity constraint that requires that every value in a column or set of columns be unique.

**universal rowid**

A data type that can store all types of rowids. Oracle uses universal rowids to store the addresses of index-organized and non-Oracle tables.

**unplugged PDB**

A self-contained set of PDB data files, and an XML metadata file that specifies the locations of the PDB files.

**unusable index**

An index that is not maintained by DML operations and which the optimizer ignores. All indexes are usable (default) or unusable.

**updatable join view**

A view that is defined on two or more base tables or views and permits DML operations.

**user event trigger**

An event trigger that is fired because of events related to user logon and logoff, DDL statements, and DML statements.

**user global area (UGA)**

See UGA.

**user name**

The name by which a user is known to Oracle Database and to other users. Every user name is associated with a password, and both must be entered to connect to Oracle Database.

**user privilege**

The right to run specific SQL statements.

**user process**

See client process.

**user profile**

A named set of resource limits and password parameters that restrict database usage and database instance resources for a user.

**view**

A custom-tailored presentation of the data in one or more tables. The views do not actually contain or store data, but derive it from the tables on which they are based.

**virtual column**

A column that is not stored on disk. The database derives the values in virtual columns on demand by computing a set of expressions or functions.

**virtual full backup**

A complete database image as of one distinct point in time, maintained efficiently by a Recovery Appliance through the indexing of incremental backups from a protected database. The virtual full backups contain individual blocks from multiple incremental backups. For example, if you take a level 0 backup on Monday with SCN 10000, and if you take an

incremental level 1 backup on Tuesday with SCN 11000, then the Recovery Appliance metadata database shows a virtual level 0 backup current to SCN 11000.

**warehouse compression**

Hybrid Columnar Compression specified with `COLUMN STORE COMPRESS FOR QUERY`. This type of compression is useful in data warehouses.

**whole database backup**

A backup of the control file and all data files that belong to a database.

**work area**

A private allocation of PGA memory used for memory-intensive operations.

**write-ahead protocol**

The protocol that mandates that before the database writer process can write a dirty buffer, the database must write to disk the redo records associated with changes to the buffer.

**zone**

Within a zone map, a zone is a set of contiguous data blocks that stores the minimum and maximum values of relevant columns.

**zone map**

Within an attribute-clustered table, a zone map is an independent access structure that divides data blocks into zones.

# Index

## A


access drivers, external table, *4-55*
access paths, data, *5-3*, *5-28*, *9-13*, *9-17*
accounts, user, *8-1*
ACID properties, *12-1*
active transactions, *12-10*
administrative accounts, *4-7*, *8-3*
administrator privileges, *4-7*, *15-10*, *18-21*
ADR, *15-28*
advanced index compression, *5-20*
alert logs, *15-31*, *17-18*
ALL_ data dictionary views, *8-8*
ALTER DATABASE statement
  application roots, *3-5*
  RECOVER clause, *3-16*
ALTER TABLE statement, *4-18*
anonymous PL/SQL blocks, *10-3*
ANSI/ISO standard, *9-3*
APIs (application program interfaces), *18-22*
  embedded SQL statements, *9-12*
  external tables, *4-55*
  Java, *10-16*
  JDBC, *9-12*, *12-21*
  network services, *18-8*
  ODBC, *9-12*
application and networking architecture, *1-19*, *18-1*
application common objects, *3-6*, *4-10*
  creation, *3-9*
  data-linked, *2-9*, *3-11*
  extended data-linked objects, *3-12*
  metadata links, *3-11*
  metadata-linked common objects, *3-9*
  naming rules, *3-7*
application containers
  about, *3-1*
  application common objects, *3-6*, *3-7*, *3-9*, *4-10*
  application PDBs, *3-6*
  application roots, *3-5*
  application seeds, *3-6*
  container maps, *3-13*
  naming rules, *2-8*
  purpose, *3-2*–*3-4*
Application Continuity, *12-19*
  architecture, *12-21*
  benefits, *12-19*
  planned maintenance, *12-20*
  use case, *12-19*
application domain indexes, *5-29*
application PDBs, *3-6*
  naming rules, *2-8*
  synchronization, *3-13*
application processes, *17-5*
application program interface
  *See* API
application roots, *3-5*
application seeds, *2-6*, *3-6*
application servers, *1-19*, *10-17*
  about, *18-5*
applications
  in a CDB
    metadata-linked common objects, *3-9*
archived redo log files, *13-16*, *13-21*
ARCHIVELOG mode, *17-21*
archiver process (ARC*n*), *17-21*
ascending indexes, *5-17*
ASSM tablespace, *14-6*
atomicity, statement-level, *12-6*
attribute-clustered tables
  benefits, *4-44*
  dimensional hierarchies, *4-50*
  interleaved ordering, *4-50*
  join attribute clustering, *4-45*
  linear ordering, *4-49*
  overview, *4-43*
  zone maps, *4-45*, *4-46*
AUDIT statement, *9-4*
auditing, *8-1*, *8-9*, *9-4*, *10-22*, *13-4*, *15-13*
  unified audit trail, *18-5*
authentication, database, *9-11*, *17-6*
automated indexes, *5-3*, *5-8*
automatic big table caching, *16-14*, *16-17*
Automatic Database Diagnostic Monitor
  *See* ADDM
Automatic Diagnostic Repository (ADR), *15-28*, *15-31*
automatic segment space management (ASSM), *14-6*
automatic undo management, *14-33*, *14-46*

## B


B-tree indexes, *4-30*, *4-36*, *5-8*
  branch level, *5-10*
  height, *5-10*
  prefix compression, *5-18*
  reverse key, *5-17*
background processes, *1-17*, *17-11*
  mandatory, *17-12*
  optional, *17-20*
  PMAN, *17-15*
big table cache, *16-17*
BINARY_DOUBLE data type, *4-23*
BINARY_FLOAT data type, *4-23*
bitmap indexes, *5-21*
  bitmap joins, *5-24*
  locks, *5-21*
  mapping table, *5-36*
  single-table, *5-22*
  storage, *5-26*
bitmap tablespace management, *14-5*
blocking transactions, *11-10*
blocks, data
  *See* data blocks
BOOLEAN data type, *4-20*, *5-12*, *9-7*, *10-24*
branch blocks, index, *5-10*
buffer cache, database
  *See* database buffer cache
buffers
  *See* database buffers
business rules, enforcing, *7-1*, *7-2*


## C


cache fusion, *11-4*
cardinality, column, *5-21*, *9-16*
Cartesian joins, *9-7*
cartridges, *5-29*
cascading deletions, *7-11*
catalog.sql script, *8-13*
CDBs, *1-10*, *13-3*
  application common objects, *3-6*, *3-7*, *3-9*
  application containers, *3-1*–*3-4*
    application common objects, *2-9*
  application PDBs, *3-6*
  application seeds, *3-6*
  character sets, *2-4*
  common objects, *2-9*, *4-10*
  common users, *4-2*, *4-4*
    naming rules, *3-7*
  container data objects, *8-5*
  container maps, *3-13*
  containers, *2-1*
  cross-container operations, *3-16*
  current container, *17-6*
  data links, *8-10*
CDBs *(continued)*
  files, *14-41*
  local users, *4-2*, *4-8*
  metadata links, *8-10*
  root container, *2-4*
  seed PDBs, *2-6*
  services, *18-11*
  system container, *2-4*
  temp files, *14-41*
  undo mode, *14-41*
chaining, rows
  *See* row chaining
CHAR data type, *4-21*
character data types, *4-21*
  VARCHAR2, *4-21*
character sets, *4-21*
  ASCII, *4-21*
  EBCDIC, *4-21*
  Unicode, *4-21*
check constraints, *7-3*, *7-12*
checkpoint process (CKPT), *17-18*
checkpoints
  control files, *13-14*
  database shutdowns, *15-15*
  definition, *15-18*
  incremental, *15-18*
  position, *15-22*
  thread, *15-18*
cleanup helper processes (CL*nn*), *17-14*
cleanup main process (CLMN), *17-14*
client processes, *1-17*, *17-5*
  connections and, *17-6*
  sessions and, *17-6*
  shared server processes and, *18-20*
client result cache, *16-28*
client-side programming, *10-1*
client/server architecture, *18-2*
  advantages, *18-3*
CLMN background process, *17-14*
CL*nn* background processes, *17-14*
cluster indexes, *4-36*
clusters, table, *4-39*
  cluster keys, *4-35*
  hash, *4-39*
  index, *4-39*
Codd, E. F., *1-2*
code points, *4-21*
collections, PL/SQL, *10-12*
columns
  cardinality, *5-21*, *9-16*
  definition, *1-5*, *4-16*
  invisible, *4-17*
  multiple indexes on, *5-5*
  order of, *4-28*
  prohibiting nulls in, *7-4*
  virtual, *4-17*, *4-28*, *5-28*

COMMENT statement, 9-4
COMMIT statement, 9-10
committing transactions
  COMMIT statement, 9-10
  defined, 12-1
  ending transactions, 12-4
  fast commit, 17-17
  group commits, 17-17
  implementation, 17-17
  implicit commits, 9-4
  lost commit problem, 12-16
  two-phase commit, 12-24
common objects, 4-10
common user accounts, 2-1, 4-2
  naming rules, 3-7
common users, 4-4
commonality, principles of, 3-7
compiled PL/SQL
  pseudocode, 10-28
  shared pool, 10-13
  triggers, 10-28
complete refresh, 6-30
composite indexes, 5-5
composite partitioning, 6-9
compound triggers, 10-23
compression
  advanced index, 5-20
  archive, 4-33
  basic table, 4-31
  data block, 14-15
  Hybrid Columnar Compression, 4-32, 4-33
  index, 5-18
  OLTP table, 4-31
  prefix index, 5-18
  table, 4-30, 6-11
  units of Hybrid Columnar Compression, 4-33
  warehouse, 4-33
concatenated indexes, 5-5
concurrency
  definition, 11-1
  dirty reads, 11-7
  fuzzy reads, 11-7
  phantom reads, 11-7
  row locks, 11-27
  transaction isolation, 11-7, 11-12, 11-16
conditions, SQL, 9-2, 9-7
conflicting writes, 11-10
connections, client/server
  administrator privileges, 15-10
  definition, 17-6
  embedded SQL, 9-12
  listener process, 1-19, 18-8
  sessions contrasted with, 17-6
consistency
  conflicting writes, 11-10
  definition, 11-1
consistency *(continued)*
  locks, 11-7, 11-37
  multiversioning, 11-1, 11-2
constraints, integrity
  check, 7-3, 7-12
  default values, 7-16
  deferrable, 7-6, 7-15, 9-10
  enabling and disabling, 7-13
  enforced with indexes, 7-6
  foreign key, 7-3, 7-7, 7-12
  mechanisms of enforcement, 7-16
  nondeferrable, 7-15
  NOT NULL, 4-18, 7-3, 7-4
  primary key, 4-18, 7-3, 7-6
  REF, 7-3
  referential, 7-11
  self-referential, 7-10
  state of, 7-13
  unique key, 7-3, 7-5
  updates of foreign and parent keys, 7-17
  validating, 7-13, 7-14
container data objects, 8-5
container maps, 3-13
CONTAINERS_DEFAULT_TARGET property, 3-16
containers, CDB, 2-1
  root, 2-4
contention
  deadlocks, 11-23
contexts, 4-10
control files, 1-16, 13-14
  changes recorded, 13-14
  checkpoints and, 13-14
  contents, 13-14
  multiplexed, 13-15
  overview, 13-14
  used in mounting database, 15-12
cost-based optimizer
  *See* optimizer
CREATE CLUSTER statement, 4-36
CREATE DIMENSION statement, 6-35
CREATE GLOBAL TEMPORARY TABLE statement, 4-53
CREATE INDEX statement, 4-53, 5-4, 5-5, 5-8, 5-17, 5-27
CREATE MATERIALIZED VIEW statement, 6-27
CREATE PLUGGABLE DATABASE statement, 2-6
  application containers, 3-1
  AS PROXY clause, 2-7
CREATE SEQUENCE statement, 6-33
CREATE SYNONYM statement, 6-36
CREATE TABLE statement, 4-15, 4-18, 4-28
  storage parameters, 14-30
CREATE UNIQUE INDEX statement, 7-6

CREATE USER statement
temporary segments, 14-32
cross-container operations, 3-16
current container, 17-6
cursors
embedded SQL, 9-12
explicit, 10-11
fetching rows, 9-12

## D

data access languages, 1-6
data blocks, 1-17, 14-1, 14-8, 14-22, 17-16
cached in memory, 16-19
clustered rows, 4-35
coalescing free space in blocks, 14-18
compression, 14-15
format, 14-10, 14-12
locks stored in, 11-29
overview, 14-2
shown in rowids, 14-13
stored in the buffer cache, 16-11
writing to disk, 16-19
data concurrency
definition, 1-9
data consistency, 1-9
data conversion
program interface, 18-22
data dictionary, 4-7, 4-27
ALL_ prefixed views, 8-8
cache, 16-24
comments in, 9-4
content, 8-3, 16-27
DBA_ prefixed views, 8-7
dictionary managed tablespaces, 14-8
DUAL table, 8-9
how database uses, 8-9
locks, 11-34
overview, 8-1
public synonyms, 8-11
storage in a CDB, 8-12
stored subprograms, 10-5
USER_ prefixed views, 8-8
data dictionary cache, 8-12, 16-24, 16-27
data failures, protecting against human errors, 12-15
data files, 1-16
contents, 13-13
data file 1, 14-45
moving online, 13-12
named in control files, 13-14
shown in rowids, 14-13
SYSTEM tablespace, 14-45
temporary, 13-12
data integrity, 7-1
enforcing, 7-1, 8-9
data integrity *(continued)*
SQL and, 9-1
data links, 3-11, 8-10
data manipulation language
*See* DML
data object number, extended rowid, 14-13
data segments, 14-30
data types
BOOLEAN, 4-20, 5-12, 9-7, 10-24
built-in, 4-20
character, 4-21
composite types, 4-20
conversions by program interface, 18-22
DATE, 4-24
datetime, 4-24
definition, 4-20
format models, 4-24
how they relate to tables, 4-16
in PL/SQL, 4-20
NCHAR, 4-22
NUMBER, 4-23
numeric, 4-23
NVARCHAR2, 4-22
object, 4-59
reference types, 4-20
ROWID, 4-25, 4-26
TIMESTAMP, 4-25
UROWID, 4-25
user-defined, 4-20, 6-26
data warehouses
bitmap indexes in, 5-21
dimension tables, 6-35
dimensions, 6-34
materialized views, 6-27
partitioning in, 6-2
summaries, 6-27
data-linked application common objects, 3-9
data-linked common objects, 2-9, 3-6, 3-11
database applications, 1-2
database authentication, 9-11, 17-6
database buffer cache, 4-30, 14-8, 16-11, 17-16
cache hits and misses, 16-15
caching of comments, 8-12
force full database caching mode, 16-21
database buffers, 14-8
after committing transactions, 12-13
buffer bodies in flash cache, 16-15
buffer cache, 16-11
checkpoint position, 17-16
committing transactions, 17-17
definition, 16-11
writing, 17-16
database instances, 1-10, 15-1
duration, 15-5
read-only, 15-4
read/write, 15-4

database instances *(continued)*
*See also* instances, database
database links, PDBs, *2-9*
database objects, *1-5*
metadata, *8-14*
database operations, *17-8*
database resident connection pooling, *18-21*
database services, *18-12*, *18-13*
in a CDB, *18-12*
PDBs, *18-11*
database structures
control files, *13-14*
data blocks, *14-1*, *14-8*, *14-22*
data files, *13-1*
extents, *14-1*
processes, *17-1*
segments, *14-1*, *14-29*
tablespaces, *13-1*, *14-41*
database writer process (DBW), *17-16*
multiple DBW*n* processes, *17-16*
database writer process (DBW*n*)
checkpoints, *17-16*
defined, *17-16*
least recently used algorithm (LRU), *16-19*
write-ahead, *17-17*
databases
administrative accounts, *4-7*
closing, *15-16*
terminating the instance, *15-17*
definition, *1-1*, *1-10*
distributed
changing global database name, *16-27*
hierarchical, *1-2*
history, *1-3*
introduction, *1-1*
mounting, *15-12*
multiprocess, *17-4*
multiuser, *17-4*
name stored in control files, *13-14*
network, *1-2*
object-relational, *1-3*
opening, *15-12*, *15-13*
relational, *1-2*, *1-3*, *9-1*
shutting down, *15-14*
starting up, *4-7*, *15-1*
forced, *15-17*
structures
control files, *13-14*
data blocks, *14-1*, *14-8*, *14-22*
data files, *13-1*
extents, *14-1*, *14-25*
logical, *14-1*
processes, *17-1*
segments, *14-1*, *14-29*
tablespaces, *13-1*, *14-41*
DATE data type, *4-24*
datetime data types, *4-24*
DBA_ views, *8-7*
DBMS (database management system), *1-2*
DBMS_METADATA package, *8-14*
DBMS_SERVICE package, *12-20*
DBMS_SPACE_ADMIN package, *14-30*
DBMS_SQL_MONITOR package, *17-8*
DBMS_STATS package, *9-18*
DBW background process, *17-16*
DDL (data definition language), *8-1*
described, *9-4*
locks, *11-34*
deadlocks, *9-21*
definition, *11-23*
decision support systems (DSS)
materialized views, *6-27*
default values
effect of constraints, *7-16*
deferred inserts, *11-6*, *16-32*
definer's rights, *10-4*
DELETE statement, *9-6*
freeing space in data blocks, *14-17*
deletions, cascading, *7-11*
denormalized tables, *6-35*
dependencies, schema object, *4-12*
descending indexes, *5-17*
dictionary cache locks, *11-36*
dictionary managed tablespaces, *14-8*
dimension tables, *6-35*
dimensions, *6-34*
attribute-clustered tables, *4-43*, *4-44*, *4-50*
attributes, *6-35*
hierarchies, *6-35*
join key, *6-35*
normalized or denormalized tables, *6-35*
tables, *6-35*
direct path loads
Hybrid Columnar Compression, *4-34*
directory objects, *4-10*
dirty reads, *11-2*, *11-7*
disk space
data files used to allocate, *13-13*
dispatcher processes (D*nnn*)
client processes connect through Oracle Net Services, *18-17*, *18-19*
network protocols and, *18-19*
prevent startup and shutdown, *18-21*
response queue and, *18-18*
distributed databases
client/server architectures and, *18-2*
job queue processes, *17-21*
recoverer process (RECO) and, *17-19*
server can also be client in, *18-2*
transactions, *12-23*
distributed transactions, *12-9*, *12-23*
in-doubt, *12-24*

distributed transactions *(continued)*
naming, *12-9*
two-phase commit and, *12-23*
DML (data manipulation language)
indexed columns, *5-21*
invisible indexes, *5-4*
locks, *11-25*
overview, *9-6*
referential actions, *7-11*
triggers, *10-22*
drivers, *18-23*
DUAL table, *8-9*
dynamic partitioning, *17-26*
dynamic performance views, *8-12*, *8-13*
database object metadata, *8-14*
storage, *8-14*
dynamic SQL
DBMS_SQL package, *10-11*
embedded, *10-11*

## E

embedded SQL, *9-1*, *9-12*, *10-20*
enqueued transactions, *12-11*
Enterprise Manager, *15-9*, *15-17*
alert log, *15-31*
dynamic performance views usage, *8-12*
executing a package, *10-9*
lock and latch monitors, *11-35*
shutdown, *15-15*, *15-17*
equijoins, *5-24*
errors, recoverable, *12-15*
exceptions, PL/SQL, *10-11*
exclusive locks, *11-21*
row locks (TX), *11-26*
table locks (TM), *11-29*
EXECUTE statement, *10-6*
execution plans, *6-32*, *9-13*, *9-16*
EXPLAIN PLAN, *9-6*
EXPLAIN PLAN statement, *9-6*
explicit locking, *11-37*
expressions, SQL, *5-4*, *9-7*
extended data-linked objects, *3-12*
extents, *1-17*
as collections of data blocks, *14-25*
deallocation, *14-27*
defined, *14-2*
dictionary managed, *14-8*
incremental, *14-26*
locally managed, *14-5*
overview of, *14-25*
external procedures, *10-13*
external tables, *4-15*, *4-53*, *4-56*
purpose, *4-54*

## F

fact tables, *6-35*
failures
database buffers and, *15-20*
statement and process, *17-13*
fast commit, *17-17*
fast full index scans, *5-12*
fast refresh, *6-30*
fields, *4-18*
file management locks, *11-36*
files
alert log, *17-18*
initialization parameter, *15-11*, *15-23*
password
administrator privileges, *15-10*
server parameter, *15-11*, *15-23*
trace files, *17-18*
fixed SGA, *16-35*
flash cache
buffer reads, *16-15*
optimized physical reads, *16-15*
floating-point numbers, *4-23*
force full database caching mode, *16-21*
foreign keys, *4-18*, *7-7*
changes in parent key values, *7-11*
composite, *7-7*
constraints, *7-3*
indexing, *5-4*, *7-12*
nulls, *7-10*
updating parent key tables, *7-11*
updating tables, *11-30*
format models, data type, *4-24*, *4-27*
free space
automatic segment space management, *14-16*
managing, *14-16*
full index scans, *5-11*
full table scans, *5-3*, *5-12*, *9-17*
default mode, *16-19*
LRU algorithm and, *16-19*
parallel exe, *17-26*
function-based indexes, *5-27*, *5-28*
functions, *9-7*
function-based indexes, *5-27*
hash, *6-7*
PL/SQL, *10-3*
SQL, *4-26*
fuzzy reads, *11-7*

## G

GDSCTL utility, *12-20*
global database names
shared pool and, *16-27*
global indexes, *6-12*, *6-15*

GRANT statement, *6-36*, *9-4*
group commits, *17-17*

## H

handles for SQL statements, *16-6*
hash clusters, *4-39*
  cluster keys, *4-40*
  hash collisions, *4-42*
  hash key values, *4-40*
  queries, *4-40*
  single-table, *4-41*
  storage, *4-42*
hash functions, *4-39*, *6-7*
hash index, *16-36*
hash partitioning, *6-7*
headers, data block, *11-29*, *14-11*
heap-organized tables, *4-11*, *5-30*
height, index, *5-10*
hierarchies
  join key, *6-35*
  levels, *6-35*
hints, optimizer, *9-13*, *9-19*, *16-32*
Hybrid Columnar Compression, *4-32*, *4-33*
  compression units, *4-33*
  DML, *4-34*
  row-level locks, *4-33*

## I

IM column store
  *See* In-Memory Column Store
IM space manager
  *See* In-Memory Space Manager
immutable tables
  about, *4-59*
in-flight transactions, *12-15*
In-Memory Column Store, *1-18*, *4-32*, *13-3*
  SQL execution, *9-23*
in-place refresh method, *6-31*
incremental refresh, *6-30*
index clustering factor, *5-15*
index compression, *5-18*
index unique scans, *5-13*
index-organized tables, *5-30*, *5-34*
  benefits, *5-30*
  bitmap indexes, *5-36*
  characteristics, *5-31*
  row overflow, *5-33*
  secondary indexes, *5-34*
indexes
  advanced index compression, *5-20*
  application domain, *5-29*
  ascending, *5-17*
  automated, *5-3*, *5-8*
  B-tree, *4-30*, *5-8*
indexes *(continued)*
  benefits, *5-3*
  bitmap, *5-21*, *5-22*, *5-26*, *5-36*
  bitmap join, *5-24*
  branch blocks, *5-10*
  cardinality, *5-21*
  composite, *5-5*
  compressed, *5-18*
  concatenated, *5-5*
  definition, *1-6*, *4-10*
  descending, *5-17*
  domain, *5-29*
  enforcing integrity constraints, *7-6*
  extensible, *5-29*
  fast full scans, *5-12*
  function-based, *5-27*, *5-28*
  global, *6-12*, *6-15*
  index clustering factor, *5-15*
  invisible, *5-4*
  keys, *5-4*, *7-6*
  leaf blocks, *5-10*
  local, *6-12*
  local partitioned, *6-13*
  multiple, *5-5*
  nonprefixed, local, *6-14*
  nonunique, *5-6*
  overview, *5-1*
  partitioned, *6-12*, *6-15*, *6-18*
  partitioning
    index, *6-15*
  prefixed, local, *6-14*
  prefixes, *5-18*
  range scans, *5-12*
  reverse key, *5-17*
  scans, *5-10*, *5-14*, *9-17*
  secondary, *5-34*
  segments, *5-8*, *5-26*
  selectivity, *5-12*
  storage, *5-8*, *6-15*
  storage space, *6-18*
  types, *5-6*
  unique, *5-6*
  unusable, *5-4*
indexes, updates, *5-8*
information systems, *1-1*
INIT.ORA
  *See* initialization parameter file.
initialization parameter file, *15-11*, *15-23*
  startup, *15-11*
initialization parameters
  about, *15-24*
  basic, *15-23*
  OPEN_CURSORS, *16-6*
  SERVICE_NAMES, *18-15*
inner joins, *9-7*
INSERT statement, *9-6*

instances, database, *1-10*, *9-11*, *15-1*
associating with databases, *15-12*
duration, *15-5*
failure, *13-17*
failures, *11-26*
memory structures of, *16-1*
process structure, *17-1*, *17-4*
recovery of
SMON process, *17-16*
service names, *18-8*
shutting down, *15-14*, *15-17*
terminating, *15-17*
INSTEAD OF triggers, *10-22*
integrity constraints, *7-1*
advantages, *7-1*, *7-2*
check, *7-12*
definition, *4-27*
nondeferrable, *7-15*
updates of foreign and parent keys, *7-17*
validating, *7-14*
views, *6-24*
internal locks, *11-36*
internal tables, *4-55*
interval partitioned tables, *6-5*
invisible columns, *4-17*
invisible indexes, *5-4*
invoker's rights, *10-4*
isolation levels
serialization, *11-12*
isolation levels, transaction, *11-7*
read committed, *11-9*
read-only, *11-16*
setting, *11-37*

## J

Java
overview, *10-14*
SQLJ translator, *10-20*
stored procedures, *1-7*, *10-19*
virtual machine, *10-16*
JDBC
accessing SQL, *10-19*
driver types, *10-20*
drivers, *10-20*
embedded SQL, *9-12*
job queue processes, *17-21*
jobs, *17-1*
join attribute clustering, *4-45*
joins
views, *6-25*
joins, table, *5-24*, *8-3*, *9-7*
Cartesian, *9-7*
clustered tables, *4-35*
conditions, *5-24*
inner joins, *9-7*
joins, table *(continued)*
join conditions, *9-7*
outer joins, *9-7*
views, *6-22*

## K

key compression
*See* prefix compression
keys
concatenation of index, *5-18*
foreign, *7-7*, *7-12*
indexes, *5-4*, *5-18*, *7-6*
natural, *7-6*
parent, *7-7*, *7-12*
partition, *6-3*
prefixed index, *5-10*
referenced, *7-7*, *7-12*
reverse, *5-17*
surrogate, *7-6*
unique, *7-5*
values, *7-3*

## L

large pool, *16-31*
deferred inserts, *16-32*
memory management, *16-32*
latches
definition, *11-35*
leaf blocks, index, *5-10*
least recently used (LRU) algorithm
database buffers and, *16-14*
full table scans and, *16-19*
latches, *16-19*
shared SQL pool, *16-27*, *16-32*
LGWR background process, *17-17*
library cache, *16-24*, *16-25*
list partitioning, *6-6*
listener process, *1-19*, *18-8*
listener registration process (LREG), *17-15*
service names, *18-8*
listener registration process (LREG), *17-15*
listeners, *1-19*, *18-8*
listener registration process (LREG), *17-15*
service names, *18-8*
local indexes, *6-12*
local partitioned indexes, *6-13*
local temporary tablespaces, *14-51*
local users, *4-2*, *4-8*
locally managed tablespaces, *14-5*
LOCK TABLE statement, *9-6*
locks, *11-7*
after committing transactions, *12-13*
automatic, *11-18*, *11-24*
bitmap indexes, *5-21*

locks *(continued)*
conversion, *11-21*, *11-22*
deadlocks, *9-21*, *11-23*
definition, *1-9*
dictionary, *11-34*
dictionary cache, *11-36*
DML, *11-25*
duration, *11-18*, *11-22*
escalation, *11-21*
exclusive, *11-21*
exclusive DDL, *11-34*
exclusive table, *11-29*
file management locks, *11-36*
Hybrid Columnar Compression, *4-33*
latches, *11-35*
log management locks, *11-36*
manual, *11-37*
overview of, *11-7*
parse, *11-35*
restrictiveness, *11-21*
rollback segments, *11-36*
row (TX), *11-26*
row exclusive table, *11-29*
row share, *11-29*
share DDL, *11-34*
share locks, *11-21*
share row exclusive, *11-29*
share table, *11-29*
system, *11-35*
table, *5-3*, *9-6*
table (TM), *11-29*
tablespace, *11-36*
types of, *11-24*
unindexed foreign keys and, *11-30*
user-defined, *11-38*
log management locks, *11-36*
log switches
archiver process, *17-21*
log sequence numbers, *13-18*
log writer process (LGWR), *17-17*
group commits, *17-17*
online redo logs available for use, *13-18*
redo log buffers and, *16-23*
write-ahead, *17-17*
writing to online redo log files, *13-18*
log-based refresh, *6-30*
logical database structures
data blocks, *1-17*
definition, *1-17*
extents, *1-17*
segments, *1-17*
tablespaces, *1-17*, *14-41*
logical rowids, *5-34*
logical transaction IDs, *12-17*
LONG data type
storage of, *4-28*
lost updates, *1-9*, *11-10*
lost writes
shadow tablespaces, *14-48*
LRU, *16-14*, *16-19*
shared SQL pool, *16-27*

## M

manual locking, *11-37*
mapping tables, *5-36*
master tables, *6-27*
materialized views, *6-27*
characteristics, *6-29*
complete refresh, *6-30*
fast refresh, *6-30*
in-place and out-of-place refresh, *6-31*
incremental refresh, *6-30*
refresh
job queue processes, *17-21*
refreshing, *6-30*, *6-31*
zone maps and, *4-46*
memoptimize pool, *16-36*
memory
allocation for SQL statements, *16-25*
content of, *16-1*
processes use of, *17-1*
stored procedures, *10-4*
MERGE statement, *9-6*
metadata links, *3-11*, *8-10*
metadata-linked application common objects, *3-9*
metadata-linked common objects, *3-6*, *3-9*
metadata links, *3-11*
metrics, *8-12*
multiblock writes, *17-16*
multiplexing
control files, *13-15*
redo log file groups, *13-20*
redo log files, *13-20*
multitenant architecture
definition, *1-10*
multitenant container databases
*See* CDBs
multiversion read consistency, *8-14*, *11-1*, *11-2*, *11-4*, *11-7*, *14-33*, *14-37*
deferred inserts, *11-6*
dirty reads, *11-2*
read committed isolation level, *11-9*
statement-level, *1-9*, *11-3*
transaction-level, *11-4*
undo segments, *11-4*
mutexes, *11-36*

## N

NaN (not a number), *4-23*
natural keys, *7-6*

NCHAR data type, *4-22*
network listener process
  connection requests, *18-19*
networks
  client/server architecture use of, *18-2*
  communication protocols, *18-23*, *18-24*
  dispatcher processes and, *18-19*
  drivers, *18-23*
  listener processes of, *18-8*
  Oracle Net Services, *18-7*
NLS_DATE_FORMAT parameter, *4-24*
NOAUDIT statement, *9-4*
nonunique indexes, *5-6*
normalized tables, *6-35*
NOT NULL constraints, *7-3*, *7-4*
nulls, *4-18*
  foreign keys, *7-10*
  how stored, *4-18*, *4-30*
  indexed, *5-6*
  prohibiting, *7-4*
NUMBER data type, *4-23*
numbers, floating point, *4-23*
numeric data types, *4-23*
  floating-point numbers, *4-23*
NVARCHAR2 data type, *4-22*

## O

object tables, *4-15*, *4-59*
object types, *4-59*, *6-26*
object views, *6-26*
OLAP
  index-organized tables, *5-30*
OLTP
  table compression, *4-31*
online redo log, *13-17*, *14-37*
  archiver process (ARC*n*), *17-21*
  buffer management, *17-17*
  checkpoints, *13-14*
  committed data, *15-20*
  committing a transaction, *17-17*
  log switch
    archiver process, *17-21*
  log writer process, *16-23*, *17-17*
  rolling forward, *15-20*
  undo data in, *14-37*
  writing buffers, *17-17*
online redo log files, *1-16*
OPEN_CURSORS parameter
  managing private SQL areas, *16-6*
operating systems
  communications software, *18-24*
  privileges for administrator, *15-10*
optimized physical reads, *16-15*
optimizer, *9-2*, *9-12*
  adaptive optimization, *9-16*
optimizer *(continued)*
  components, *9-14*
  estimator, *9-16*
  execution, *9-23*
  execution plans, *6-32*, *9-13*, *9-16*, *9-22*
  function-based indexes, *5-28*
  hints, *9-13*, *9-19*, *16-32*
  invisible indexes, *5-4*
  partitions in query plans, *6-2*
  plan generator, *9-16*
  query plans, *9-22*
  query transformer, *6-32*, *9-16*
  row sources, *9-22*
  statistics, *4-28*, *9-18*, *9-22*
Oracle ASM (Automatic Storage Management), *13-4*, *13-5*
Oracle Automatic Storage Management
  *See* Oracle ASM
Oracle blocks, *14-2*
Oracle Call Interface
  *See* OCI
Oracle code, *18-22*
Oracle Data Pump, *14-44*
Oracle Database
  history, *1-3*
Oracle Enterprise Manager
  *See* Enterprise Manager
Oracle Flashback Technology, *1-9*
Oracle Internet Directory, *18-15*
Oracle JVM
  main components, *10-17*
  overview, *10-16*
Oracle Net, *1-19*
Oracle Net Listener, *1-19*
Oracle Net Services, *1-19*, *18-7*
  client/server systems use of, *18-7*
  overview, *18-7*
  shared server requirement, *18-19*
Oracle Net Services Connection Manager, *10-14*
Oracle processes, *17-4*, *17-9*
Oracle program interface (OPI), *18-23*
Oracle RAC
  *See* Oracle Real Application Clusters
Oracle Real Application Clusters, *13-4*
  reverse key indexes, *5-17*
Oracle Sharding
  about, *6-19*
  architecture, *1-14*
Oracle XA
  session memory in the large pool, *16-31*
ORDBMS (object-relational database management system), *1-3*
out-of-place refresh method, *6-31*
outer joins, *9-7*

## P


packages, *10-7*
  advantages of, *10-8*
  creation, *10-9*
  executing, *10-13*
  for locking, *11-38*
  private, *10-8*
  public, *10-8*
  shared SQL areas and, *16-27*
  subprogram executions, *10-9*
pages, *14-2*
parallel execution, *17-24*
  coordinator, *17-26*
  server, *17-26*
  servers, *17-26*
  tuning, *17-24*
parallel SQL, *17-24*
  coordinator process, *17-26*
  server processes, *17-26*
parameters
  initialization, *15-23*
    locking behavior, *11-24*
parse locks, *11-35*
parsing, SQL, *9-20*, *9-21*
  embedded SQL, *9-12*
  hard parse, *11-35*
  storage of information, *8-12*
partial global partitioned indexes, *6-18*
partitioned change tracking refresh, *6-30*
partitioned indexes, *6-13*
  global, *6-15*
  storage, *6-15*
partitioned tables, *6-7*
  interval, *6-5*
  reference, *6-8*
partitioning
  by hash, *6-7*
  by interval, *6-5*
  by list, *6-6*
  by range, *6-4*
  by reference, *6-8*
  composite, *6-9*
  index, *6-15*
  indexes, *6-13*
  key, *6-3*
partitions
  characteristics, *6-3*
  composite partitioning, *6-3*
  definition, *4-10*
  dynamic partitioning, *17-26*
  elimination from queries, *6-14*
  index, *6-12*
  keys, *6-3*
  materialized views, *6-29*
  overview, *6-2*
partitions *(continued)*
  partitions
    composite, *6-3*
  range, *6-3*
  segments, *14-30*
  single-level, *6-3*
  strategies, *6-3*
  table, *6-4*, *6-6*, *6-10*, *6-18*
passwords
  administrator privileges, *15-10*
  connecting with, *17-6*
PCTFREE storage parameter
  how it works, *14-16*
PDB$SEED, *2-6*
PDBs, *1-10*, *13-3*
  character sets, *2-4*
  cloning, *2-7*
  common users, *3-7*
  connecting to, *18-13*
  containers, *2-1*
  current container, *17-6*
  database links, *2-9*
  definition, *2-5*
  naming rules, *2-8*
  proxy, *2-6*, *2-7*
  purpose of, *2-7*
  services, *18-11*–*18-13*
  temp files, *14-41*
  types, *2-6*
performance
  group commits, *17-17*
  packages, *10-8*
PGA
  *See* program global area (PGA)
phantom reads, *11-7*
physical database structures
  control files, *1-16*, *13-14*
  data files, *1-16*
  online redo log files, *1-16*
physical guesses, *5-35*
PL/SQL
  anonymous blocks, *10-3*, *10-10*
  collections, *10-12*
  data types, *4-20*
  dynamic SQL, *10-11*
  exceptions, *10-11*
  execution, *10-13*
  execution of subprograms, *10-6*
  language constructs, *10-11*
  overview, *10-3*
  package creation, *10-9*
  packages, *8-14*, *10-7*, *10-8*
  PL/SQL engine, *10-13*
  program units, *10-4*, *16-27*
    shared SQL areas and, *16-27*
  records, *10-12*, *10-13*

PL/SQL *(continued)*
stored procedures, *1-7*, *8-8*, *10-3*, *10-4*, *10-8*
units, *10-3*
compiled, *10-13*
plan
SQL execution, *9-6*
pluggable databases
*See* PDBs
PMAN background process, *17-15*
PMON background process, *17-14*
pragmas, PL/SQL, *12-22*
precompilers, *10-1*
embedded SQL, *9-12*
predicates, SQL, *5-11*, *9-2*
primary key constraints, *7-3*
primary keys, *4-18*, *5-3*
constraints, *7-6*
hash clusters, *4-41*
index-organized tables, *4-15*
private SQL areas, *16-25*
described, *16-25*
how managed, *16-6*
parsing and, *9-21*
private synonyms, *6-36*
privileges, *8-1*, *9-11*
administrator, *15-10*
granting, *9-4*
PL/SQL procedures and, *10-4*
revoking, *9-4*
procedures, *10-3*, *10-5*
advantages, *10-4*
execution, *10-6*, *10-13*
execution in a PL/SQL package, *10-9*
external, *10-13*
memory allocation, *10-4*
security, *10-4*
shared SQL areas and, *16-27*
stored procedures, *1-7*, *8-8*, *10-3*, *10-13*
process manager process (PMON)
state objects, *17-15*
process monitor process (PMON)
described, *17-14*
processes, *17-1*
archiver (ARC*n*), *17-21*
background, *1-17*, *17-11*
checkpoints and, *17-16*
client, *1-17*, *17-5*
dedicated server, *18-20*
definition, *1-17*
distributed transaction resolution, *17-19*
job queue, *17-21*
listener, *18-8*
shared servers and, *18-19*
log writer (LGWR), *17-17*
multithreaded, *17-4*
Oracle, *17-9*
processes *(continued)*
parallel execution coordinator, *17-26*
parallel execution servers, *17-26*
process monitor (PMON), *17-14*
recoverer (RECO), *17-19*
server, *1-17*, *17-9*
shared, *18-19*, *18-20*
shared server, *18-17*
client requests and, *18-18*
structure, *17-1*
system monitor (SMON), *17-16*
user
recovery from failure of, *17-13*
sharing server processes, *18-19*
processing
parallel SQL, *17-24*
program global area (PGA), *1-18*, *16-2*
shared server, *18-20*
shared servers, *18-20*
program interface, *18-22*
Oracle side (OPI), *18-23*
structure of, *18-23*
user side (UPI), *18-23*
program units
shared pool and, *16-27*
programming, server-side, *1-7*, *10-1*
proxy PDBs, *2-6*, *2-7*, *2-9*
pseudocode
triggers, *10-28*
pseudocolumns, *4-26*, *5-31*
public synonyms, *6-36*

## Q

queries
blocks, *9-9*
definition, *9-7*
implicit, *11-9*
in DML, *9-6*
parallel processing, *17-24*
query blocks, *9-16*
query transformer, *9-16*
SQL language and, *9-1*
stored, *6-22*
subqueries, *6-22*, *9-6*, *9-9*
query blocks, *9-16*
query optimizer
*See* optimizer
query plans, *9-22*
partitioning and, *6-2*
query rewrite, *6-34*
query transformer, *6-32*

## R

range partitioning, *6-4*

range partitions, *6-3*
range scans, index, *5-12*
RDBMS (relational database management system), *1-3*
RDMA, *16-36*
read committed isolation, *11-9*
read consistency
  *See* multiversion read consistency
read uncommitted, *11-7*
read-only isolation level, *11-16*
Real Application Clusters
  cache fusion, *11-4*
  system monitor process and, *17-16*
  threads of online redo log, *13-17*
records, PL/SQL, *10-12*, *10-13*
recoverable errors, *12-15*
recoverer process (RECO), *17-19*
  in-doubt transactions, *12-24*
recovery
  database buffers and, *15-20*
  distributed processing in, *17-19*
  instance recovery
    SMON process, *17-16*
  media
    dispatcher processes, *18-21*
  process recovery, *17-13*
  required after terminating instance, *15-17*
  rolling back transactions, *15-22*
Recovery Manager, *8-12*
redo log buffer, *13-17*
redo log files
  available for use, *13-18*
  circular use of, *13-18*
  contents of, *13-21*
  distributed transaction information in, *13-17*
  group members, *13-20*
  groups, defined, *13-20*
  instance recovery use of, *13-17*
  LGWR and the, *13-18*
  members, *13-20*
  multiplexed, *13-20*
  online, defined, *13-17*
  redo entries, *13-21*
  threads, *13-17*
redo logs buffer, *16-23*
redo records, *13-21*
REF constraints, *7-3*
reference partitioned tables, *6-8*
referential integrity
  examples of, *7-16*
  self-referential constraints, *7-16*
refresh
  incremental, *6-30*
  job queue processes, *17-21*
  materialized views, *6-30*, *6-31*
relational database management system
  *See* RDBMS
relations, simple, *1-2*
reserved words, *9-3*
response queues, *18-18*
result cache, *16-28*
result sets, SQL, *4-15*, *4-26*, *4-51*, *6-25*, *9-7*
RESULT_CACHE clause, *16-28*
reverse key indexes, *5-17*
REVOKE statement, *9-4*
rights, definer's and invoker's, *10-4*
roles, *4-10*, *8-8*, *9-11*
rollback, *12-13*
  described, *12-13*
  ending a transaction, *12-13*
  implicit in DDL, *9-4*
  statement-level, *11-23*, *12-6*
  to a savepoint, *12-11*
  transaction-level, *9-10*
rollback segments
  locks on, *11-36*
  parallel recovery, *15-22*
  use of in recovery, *15-22*
ROLLBACK statement, *12-7*
rolling back, *12-1*, *12-13*
root container, *2-1*, *2-4*
row chaining, *14-8*, *14-20*
row data (section of data block), *14-12*
row directories, *14-11*
row locks, *11-26*
  concurrency, *11-27*
  storage, *11-29*
row pieces, *4-29*
row source generation, *9-22*
ROWID data type, *4-25*, *4-26*
rowids, *4-30*
  foreign, *4-25*
  index, *5-6*
  logical, *4-25*, *5-34*
  physical, *4-25*
  row migration, *14-20*
  scans, *9-17*
  universal, *4-25*
rows
  addresses, *4-30*
  chaining across blocks, *4-29*, *14-20*
  clustered, *4-29*
  definition, *1-5*, *4-18*
  format of in data blocks, *14-11*
  locking, *11-26*
  locks on, *11-26*
  migrating to new block, *14-20*
  row set, *9-22*
  row source, *9-22*
  shown in rowids, *14-13*
  storage, *4-29*
  triggers, *10-22*

## S


sample schemas, *4-14*
SAVEPOINT statement, *9-10*
savepoints, *9-10*, *12-10*
  definition, *12-10*
  implicit, *12-6*
  rolling back to, *12-11*
  SAVEPOINT statement, *12-7*
scans
  cluster, *9-17*
  fast full index, *5-12*
  full index, *5-11*
  full table, *5-3*, *9-17*, *16-19*
  index, *5-10*, *9-17*
  index skip, *5-14*
  range, *4-40*
  rowid, *9-17*
  unique index, *5-13*
schema objects
  definitions, *1-5*, *8-1*, *9-4*
  dependencies, *4-12*, *6-23*
  dimensions, *6-34*
  indexes, *5-1*
  introduction, *4-2*
  materialized views, *6-27*
  relationship to data files, *13-10*
  sequences, *6-33*
  storage, *4-11*
schemas, *1-5*, *4-2*
  sample, *4-14*
SCN
  *See* system change numbers
secondary indexes, *5-34*
  benefits, *5-34*
  physical guesses, *5-35*
security
  administrator privileges, *15-10*
  definer's rights, *10-4*
  program interface enforcement of, *18-22*
  views, *6-22*
seed PDB, *2-1*
segments, *1-17*, *14-29*
  data, *14-30*
  defined, *14-2*
  index, *5-8*, *5-26*
  overview of, *14-29*
  table storage, *4-28*
  temporary, *4-53*, *14-32*
    allocating, *14-32*
  user, *14-30*
select lists, SQL, *9-7*
SELECT statement, *9-6*
selectivity, *5-12*
self-referential integrity constraints, *7-10*
sequences
  characteristics, *6-33*
  concurrent access, *6-33*
  definition, *4-10*, *6-33*
  surrogate keys, *7-6*
serializability, transactional, *11-1*
serialization isolation level, *11-12*
server parameter file
  about, *15-24*
  startup, *15-11*
server processes, *1-17*, *17-9*
  creation, *17-11*
  listener process, *18-8*
server-side programming, *10-1*
  overview, *10-1*
servers
  client/server architecture, *18-2*
  shared
    architecture, *17-2*, *18-17*
    processes of, *18-17*, *18-20*
service names, *18-8*
SERVICE_NAMES parameter, *18-15*
service-oriented architecture, *1-19*, *18-6*
session control statements, *9-11*
sessions, *9-11*
  connections contrasted with, *17-6*
  defined, *17-6*
  memory allocation in the large pool, *16-31*
  sequence generation in, *6-33*
SET CONSTRAINT statement, *9-10*
SET TRANSACTION statement, *9-10*, *12-4*
SGA (system global area), *1-18*, *16-2*
  allocating, *15-11*
  big table cache, *16-17*
  contents of, *16-9*
  data dictionary cache, *8-12*, *16-27*
  database buffer cache, *16-11*
  fixed SGA, *16-35*
  large pool, *16-31*
  redo log buffer, *12-10*, *16-23*
  rollback segments and, *12-10*
  shared and writable, *16-9*
  shared pool, *10-4*, *16-24*
  variable parameters, *15-23*
shadow lost write protection, *14-47*
  configuring, *14-50*
  enabling, *14-48*
  purpose, *14-48*
shadow tablespaces, *14-47*
  creating, *14-50*
  how they work, *14-48*
  purpose, *14-48*
  user interface, *14-49*
sharded database (SDB), *6-19*
sharded tables, *6-19*
shards, *6-19*

share DDL locks, *11-34*
share locks, *11-21*
shared pool, *10-13*, *16-24*, *16-27*
  allocation of, *16-27*
  dependency management and, *16-27*
  described, *16-24*
  flushing, *16-27*
  latches, *11-35*
  parse locks, *11-35*
shared server
  described, *17-2*
  dispatcher processes, *18-19*
  Oracle Net Services or SQL*Net V2 requirement, *18-19*
  processes, *18-20*
  processes needed for, *18-17*
  restricted operations in, *18-21*
  session memory in the large pool, *16-31*
shared server processes (*Snnn*), *18-20*
  described, *18-20*
shared SQL areas, *6-24*, *16-24*, *16-25*, *16-27*
  dependency management and, *16-27*
  described, *16-25*
  parse locks, *11-35*
  procedures, packages, triggers and, *16-27*
shared temporary tablespaces, *14-51*
shutdown, *15-14*, *15-17*
  abnormal, *15-17*
  prohibited by dispatcher processes, *18-21*
  steps, *15-14*
SHUTDOWN ABORT statement, *15-17*
Simple Object Access Protocol (SOAP)
  *See* SOAP
simple triggers, *10-23*
single-level partitioning, *6-3*
SMON background process, *17-16*
SOA, *1-19*, *18-6*
SOAP (Simple Object Access Protocol), *1-19*
software code areas, *16-2*
space management
  extents, *14-25*
  PCTFREE, *14-16*
  row chaining, *14-20*
  segments, *14-29*
Space Management Coordinator Process (SMCO) background process, *16-32*
SQL, *9-1*, *9-3*
  conditions, *9-2*, *9-7*
  data definition language (DDL), *9-4*
  data manipulation language (DML), *9-6*
  definition, *1-6*
  dictionary cache locks, *11-36*
  dynamic SQL, *10-11*
  embedded, *9-1*, *9-12*
  executable, *12-4*
  execution, *9-23*, *12-6*
SQL *(continued)*
  expressions, *5-4*, *9-7*
  functions, *4-26*
  interactive, *9-1*
  memory allocation for, *16-25*
  operators, *9-2*
  optimization, *9-22*
  Oracle, *1-6*, *9-3*
  overview, *9-1*
  parallel execution, *17-24*
  parsing, *9-20*, *9-21*
  PL/SQL and, *10-3*
  predicates, *5-11*, *9-2*
  processing, *9-20*
  reserved words, *9-3*
  result sets, *4-15*, *4-26*, *4-51*, *6-25*, *9-7*
  select lists, *9-7*
  session control statements, *9-11*
  standards, *9-3*
  statements, *9-3*
  subqueries, *6-22*, *9-9*
  system control statements, *9-11*
  transaction control statements, *9-10*
  transactions, *12-1*
  types of statements, *9-3*
SQL areas
  private, *16-25*
  shared, *16-25*
SQL*Plus
  alert log, *15-31*
  executing a package, *10-9*
  lock and latch monitors, *11-35*
SQLJ standard, *10-20*
SRVCTL utility, *12-20*
standards
  ANSI/ISO, *9-3*
    isolation levels, *11-7*
star schemas, *4-50*
startup, *15-1*
  prohibited by dispatcher processes, *18-21*
statement-level atomicity, *12-6*
statement-level read consistency, *11-3*
statement-level rollback, *11-23*, *12-6*
statements, SQL, *9-3*
statistics, *4-28*, *8-12*, *9-13*, *16-27*
  definition, *9-18*
  Java-related, *16-34*
  join order, *9-7*
  optimizer, *9-22*
  undo retention, *14-47*
storage
  logical structures, *14-41*
  nulls, *4-18*
  triggers, *10-28*
stored procedures
  *See* procedures

Structured Query Language (SQL), 9-1
structures
locking, 11-34
logical, 14-1
data blocks, 14-1, 14-8, 14-22
extents, 14-1, 14-25
segments, 14-1, 14-29
tablespaces, 13-1, 14-41
physical
control files, 13-14
data files, 13-1
processes, 17-1
subprograms, PL/SQL
*See* procedures
subqueries, 6-22, 9-6, 9-9
summaries, 6-27
surrogate keys, 7-6
synonyms
constraints indirectly affect, 7-17
data dictionary views, 8-11
definition, 4-10, 6-36
private, 6-36
public, 6-36, 8-8
securability, 6-36
SYS user name, 4-7
data dictionary tables, 8-9
SYSDBA privilege, 15-10
SYSOPER privilege, 15-10
system change numbers (SCN), 12-6
committed transactions, 12-13
defined, 12-13
definition, 11-4
when assigned, 13-21
system container, 2-4
system control statements, 9-11
system global area
*See* SGA
system locks, 11-35
internal, 11-36
latches, 11-35
mutexes, 11-36
system monitor process (SMON), 17-16
defined, 17-16
Real Application Clusters and, 17-16
rolling back transactions, 15-22
SYSTEM tablespace
data dictionary stored in, 14-45
online requirement of, 14-55
SYSTEM user name, 4-7

## T

table clusters
cluster keys, 4-35
definition, 4-35
hash cluster storage, 4-42
table clusters *(continued)*
hash clusters, 4-39–4-41
indexed, 4-36
overview, 4-1
scans, 9-17
table partitions, 6-4
tables
attribute-clustered, 4-45
automatic big table caching, 16-14, 16-17
base, 6-23
characteristics, 4-18
clustered, 4-35
compression, 4-30, 6-11
definition, 1-2, 1-5, 4-10
dimension, 6-35
directories, 14-11
DUAL, 8-9
dynamic partitioning, 17-26
external, 4-53, 4-55, 4-56
fact, 6-35
full table scans, 5-3, 16-19
heap-organized, 4-11, 5-30
index-organized, 5-30, 5-31, 5-33, 5-34
integrity constraints, 7-1
joins, 5-24
master, 6-27
normalized or denormalized, 6-35
object, 4-59
overview, 4-1, 4-15
partitioned, 6-3, 6-6, 6-7
partitions, 6-10
permanent, 4-15
storage, 4-28
temporary, 4-51, 4-53, 14-33
transaction, 12-4
truncating, 9-4, 14-27
views of, 6-22
virtual, 8-14
tables, attribute-clustered, 4-43
dimensional hierarchies, 4-50
linear ordering, 4-49
zone maps, 4-45, 4-46
tables, base, 6-22
tables, external, 4-15
tables, object, 4-15
tables, temporary, 4-15
tablespaces, 1-17, 14-41
described, 14-41
dictionary managed, 14-8
locally managed, 14-5
locks on, 11-36
offline, 14-55
online, 14-55
overview of, 14-41
schema objects, 4-11
space allocation, 14-4

tablespaces *(continued)*
used for temporary segments, *14-32*
tasks, *17-1*
temp files, *13-12*
temporary segments, *4-53*, *14-33*
allocating, *14-32*
allocation for queries, *14-32*
temporary tables, *4-15*
creation, *4-53*
global, *4-51*
private, *4-51*
purpose, *4-52*
temporary tablespaces, *14-51*
creation of default, *14-53*
local, *14-51*
shared, *14-51*
threads
online redo log, *13-17*
time zones, *4-25*
in date/time columns, *4-25*
TIMESTAMP data type, *4-24*, *4-25*
TO_CHAR function, *4-27*
TO_DATE function, *4-24*, *4-27*
trace files
LGWR trace file, *17-18*
transaction control statements, *9-10*
Transaction Guard, *12-15*, *12-21*
benefits, *12-16*
how it works, *12-16*
transaction management
definition, *1-8*
transaction tables, *12-4*
reset at recovery, *17-13*
transaction-level read consistency, *11-4*
transactions, *12-1*
ACID properties, *12-1*
active, *12-10*, *14-33*
assigning system change numbers, *12-13*
autonomous, *12-22*
within a PL/SQL block, *12-22*
beginning, *12-4*
blocking, *11-10*
committing, *12-16*, *17-17*
group commits, *17-17*
conflicting writes, *11-10*
deadlocks, *11-23*, *12-6*
definition, *1-8*, *12-1*
distributed, *12-9*, *12-23*
resolving automatically, *17-19*
DML statements, *9-6*
ending, *12-4*
enqueued, *12-11*
idempotence, *12-15*
in-doubt
resolving automatically, *12-24*
in-flight, *12-15*
transactions *(continued)*
isolation levels, *11-7*, *11-12*, *11-16*
isolation of, *11-7*
logical transaction ID, *12-17*
naming, *12-9*
read consistency, *11-4*
rolling back, *12-13*
partially, *12-11*
savepoints in, *12-10*
serializability, *11-1*
setting properties, *9-10*
structure, *12-3*
terminated, *14-33*
terminating the application and, *12-4*
transaction control statements, *9-10*
transaction history, *11-6*
transaction ID, *12-1*, *12-4*
triggers, *10-1*
cascading, *10-22*
components of, *10-24*
compound, *10-23*
effect of rollbacks, *12-6*
firing (executing), *10-28*
privileges required, *10-28*
INSTEAD OF, *10-22*
overview, *10-21*
restrictions, *10-24*
row, *10-22*
shared SQL areas and, *16-27*
simple, *10-23*
statement, *10-22*
storage of, *10-28*
timing, *10-23*
UNKNOWN does not fire, *10-24*
uses of, *10-22*
TRUNCATE statement, *9-4*, *14-27*
tuples, definition, *1-2*
two-phase commit
transaction management, *12-23*

## U

undo management, automatic, *14-33*
undo mode
CDBs, *14-41*
undo retention period, *14-47*
undo segments, *12-4*, *14-33*
read consistency, *11-4*
temporary, *13-17*, *14-33*, *14-37*
undo space management
automatic undo management mode, *14-46*
undo tablespaces, *14-33*, *14-46*, *14-47*
undo retention period, *11-16*
Unicode, *4-21*
unified audit trail, *18-5*
unique indexes, *5-6*

unique key constraints, *7-3*, *7-5*
composite keys, *7-5*
NOT NULL constraints and, *7-5*
updatable join views, *6-25*
update no action constraint, *7-11*
UPDATE statement, *9-6*
updates
lost, *11-10*
tables with parent keys, *11-30*
updatability of views, *6-25*, *10-22*
updatable join views, *6-25*
upgrades
database, *8-2*, *14-45*, *14-46*
UROWID data type, *4-25*
user global area (UGA), *16-2*
user program interface (UPI), *18-23*
user segments, *14-30*
USER_ views, *8-8*
users
common, *2-1*, *3-7*, *4-4*
users, database, *4-10*
names, *8-1*
sessions and connections, *17-6*
privileges, *4-2*
temporary tablespaces, *14-32*
UTL_HTTP package, *10-7*

## V

V$ views, *8-13*
database object metadata, *8-14*
storage, *8-14*
VARCHAR2 data type, *4-21*
variables
embedded SQL, *9-12*
views, *6-22*
base tables, *6-22*
views *(continued)*
characteristics, *6-23*
constraints indirectly affect, *7-17*
data access, *6-24*
data dictionary
updatable columns, *6-25*
definition, *4-10*
DML, *6-24*
dynamic performance, *8-12*, *8-13*
indexes, *6-24*
INSTEAD OF triggers, *10-22*
integrity constraints, *6-24*
materialized, *6-27*
object, *6-26*
schema object dependencies, *6-23*
storage, *6-23*
updatability, *6-25*
uses, *6-22*
V$, *8-12*
virtual columns, *4-17*, *4-28*, *5-28*

## W

warehouse
materialized views, *6-27*
Web services, *1-19*, *18-6*
write-ahead, *17-17*

## X

XA
session memory in the large pool, *16-31*

## Z

zone maps, *4-45*, *4-46*