

Oracle® Database

Concepts

11g Release 2 (11.2)

E25789-01

September 2011

Oracle Database Concepts, 11g Release 2 (11.2)

E25789-01

Copyright © 1993, 2011, Oracle and/or its affiliates. All rights reserved.

Primary Authors: Lance Ashdown, Tom Kyte

Contributors: Drew Adams, David Austin, Vladimir Barriere, Hermann Baer, David Brower, Jonathan Creighton, Bjørn Engsig, Steve Fogel, Bill Habeck, Bill Hodak, Yong Hu, Pat Huey, Vikram Kapoor, Feroz Khan, Jonathan Klein, Sachin Kulkarni, Paul Lane, Adam Lee, Yunrui Li, Bryn Llewellyn, Rich Long, Barb Lundhild, Neil Macnaughton, Vineet Marwah, Mughees Minhas, Sheila Moore, Valarie Moore, Gopal Mulagund, Paul Needham, Gregory Pongracz, John Russell, Vivian Schupmann, Shrikanth Shankar, Cathy Shea, Susan Shepard, Jim Stenoish, Juan Tellez, Lawrence To, Randy Urbano, Badhri Varanasi, Simon Watt, Steve Wertheimer, Daniel Wong

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

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

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

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

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

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

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

Contents

Preface	xv
Audience	xv
Documentation Accessibility	xv
Related Documentation	xvi
Conventions	xvi
1 Introduction to Oracle Database	
About Relational Databases	1-1
Database Management System (DBMS)	1-1
Relational Model	1-2
Relational Database Management System (RDBMS)	1-2
Brief History of Oracle Database	1-3
Schema Objects	1-4
Tables	1-4
Indexes	1-4
Data Access	1-5
Structured Query Language (SQL)	1-5
PL/SQL and Java	1-5
Transaction Management	1-6
Transactions	1-6
Data Concurrency	1-6
Data Consistency	1-6
Oracle Database Architecture	1-7
Database and Instance	1-7
Database Storage Structures	1-8
Database Instance Structures	1-9
Application and Networking Architecture	1-10
Oracle Database Documentation Roadmap	1-12
Basic Group	1-12
Intermediate Group	1-12
Advanced Group	1-13

Part I Oracle Relational Data Structures

2 Tables and Table Clusters

Introduction to Schema Objects	2-1
Schema Object Types	2-2
Schema Object Storage	2-3
Schema Object Dependencies	2-4
SYS and SYSTEM Schemas	2-5
Sample Schemas	2-6
Overview of Tables	2-6
Columns and Rows	2-7
Example: CREATE TABLE and ALTER TABLE Statements	2-7
Oracle Data Types	2-9
Integrity Constraints	2-14
Object Tables	2-15
Temporary Tables	2-15
External Tables	2-16
Table Storage	2-18
Table Compression	2-19
Overview of Table Clusters	2-22
Overview of Indexed Clusters	2-23
Overview of Hash Clusters	2-25

3 Indexes and Index-Organized Tables

Overview of Indexes	3-1
Index Characteristics	3-2
B-Tree Indexes	3-5
Bitmap Indexes	3-13
Function-Based Indexes	3-17
Application Domain Indexes	3-19
Index Storage	3-20
Overview of Index-Organized Tables	3-20
Index-Organized Table Characteristics	3-21
Index-Organized Tables with Row Overflow Area	3-23
Secondary Indexes on Index-Organized Tables	3-23

4 Partitions, Views, and Other Schema Objects

Overview of Partitions	4-1
Partition Characteristics	4-2
Partitioned Tables	4-7
Partitioned Indexes	4-7
Partitioned Index-Organized Tables	4-12
Overview of Views	4-12
Characteristics of Views	4-13
Updatable Join Views	4-15
Object Views	4-16
Overview of Materialized Views	4-16
Characteristics of Materialized Views	4-17

Refresh Methods for Materialized Views	4-18
Query Rewrite.....	4-19
Overview of Sequences	4-20
Sequence Characteristics	4-20
Concurrent Access to Sequences.....	4-20
Overview of Dimensions	4-21
Hierarchical Structure of a Dimension.....	4-21
Creation of Dimensions.....	4-21
Overview of Synonyms	4-22

5 Data Integrity

Introduction to Data Integrity	5-1
Techniques for Guaranteeing Data Integrity	5-1
Advantages of Integrity Constraints	5-1
Types of Integrity Constraints	5-2
NOT NULL Integrity Constraints.....	5-3
Unique Constraints	5-3
Primary Key Constraints.....	5-5
Foreign Key Constraints.....	5-6
Check Constraints	5-9
States of Integrity Constraints	5-10
Checks for Modified and Existing Data	5-10
Deferrable Constraints.....	5-11
Examples of Constraint Checking	5-12

6 Data Dictionary and Dynamic Performance Views

Overview of the Data Dictionary	6-1
Contents of the Data Dictionary	6-2
Storage of the Data Dictionary	6-4
How Oracle Database Uses the Data Dictionary	6-4
Overview of the Dynamic Performance Views	6-5
Contents of the Dynamic Performance Views	6-6
Storage of the Dynamic Performance Views.....	6-6
Database Object Metadata	6-6

Part II Oracle Data Access

7 SQL

Introduction to SQL	7-1
SQL Data Access.....	7-1
SQL Standards	7-2
Overview of SQL Statements	7-3
Data Definition Language (DDL) Statements	7-3
Data Manipulation Language (DML) Statements	7-4
Transaction Control Statements	7-8
Session Control Statements.....	7-8

System Control Statement.....	7-9
Embedded SQL Statements	7-9
Overview of the Optimizer	7-10
Use of the Optimizer.....	7-10
Optimizer Components.....	7-11
Access Paths.....	7-12
Optimizer Statistics.....	7-13
Optimizer Hints.....	7-14
Overview of SQL Processing	7-15
Stages of SQL Processing	7-15
How Oracle Database Processes DML.....	7-22
How Oracle Database Processes DDL.....	7-23

8 Server-Side Programming: PL/SQL and Java

Introduction to Server-Side Programming.....	8-1
Overview of PL/SQL.....	8-2
PL/SQL Subprograms	8-3
PL/SQL Packages	8-6
PL/SQL Anonymous Blocks	8-9
PL/SQL Language Constructs.....	8-9
PL/SQL Collections and Records.....	8-10
How PL/SQL Runs.....	8-11
Overview of Java in Oracle Database.....	8-12
Overview of the Java Virtual Machine (JVM).....	8-13
Java Programming Environment	8-14
Overview of Triggers	8-16
Advantages of Triggers.....	8-17
Types of Triggers.....	8-17
Timing for Triggers.....	8-18
Creation of Triggers.....	8-18
Execution of Triggers.....	8-21
Storage of Triggers.....	8-21

Part III Oracle Transaction Management

9 Data Concurrency and Consistency

Introduction to Data Concurrency and Consistency	9-1
Multiversion Read Consistency	9-2
Locking Mechanisms	9-5
ANSI/ISO Transaction Isolation Levels	9-5
Overview of Oracle Database Transaction Isolation Levels	9-6
Read Committed Isolation Level	9-6
Serializable Isolation Level	9-8
Read-Only Isolation Level	9-11
Overview of the Oracle Database Locking Mechanism	9-11
Summary of Locking Behavior.....	9-12

Use of Locks	9-12
Lock Modes	9-15
Lock Conversion and Escalation.....	9-15
Lock Duration	9-16
Locks and Deadlocks	9-16
Overview of Automatic Locks	9-17
DML Locks	9-18
DDL Locks.....	9-24
System Locks.....	9-25
Overview of Manual Data Locks.....	9-26
Overview of User-Defined Locks.....	9-27

10 Transactions

Introduction to Transactions	10-1
Sample Transaction: Account Debit and Credit	10-2
Structure of a Transaction	10-2
Statement-Level Atomicity	10-4
System Change Numbers (SCNs).....	10-5
Overview of Transaction Control	10-6
Transaction Names	10-7
Active Transactions.....	10-7
Savepoints	10-8
Rollback of Transactions	10-10
Committing Transactions.....	10-10
Overview of Autonomous Transactions	10-11
Overview of Distributed Transactions	10-12
Two-Phase Commit	10-13
In-Doubt Transactions.....	10-13

Part IV Oracle Database Storage Structures

11 Physical Storage Structures

Introduction to Physical Storage Structures	11-1
Mechanisms for Storing Database Files	11-2
Oracle Automatic Storage Management (Oracle ASM)	11-3
Oracle Managed Files and User-Managed Files.....	11-6
Overview of Data Files.....	11-7
Use of Data Files	11-7
Permanent and Temporary Data Files	11-8
Online and Offline Data Files.....	11-9
Data File Structure	11-9
Overview of Control Files	11-10
Use of Control Files	11-10
Multiple Control Files	11-11
Control File Structure	11-11
Overview of the Online Redo Log.....	11-12

Use of the Online Redo Log.....	11-12
How Oracle Database Writes to the Online Redo Log	11-12
Structure of the Online Redo Log	11-15

12 Logical Storage Structures

Introduction to Logical Storage Structures	12-1
Logical Storage Hierarchy.....	12-2
Logical Space Management	12-2
Overview of Data Blocks	12-6
Data Blocks and Operating System Blocks.....	12-6
Data Block Format.....	12-7
Data Block Compression	12-11
Space Management in Data Blocks.....	12-11
Overview of Extents.....	12-18
Allocation of Extents	12-18
Deallocation of Extents	12-19
Storage Parameters for Extents	12-20
Overview of Segments	12-21
User Segments	12-21
Temporary Segments	12-23
Undo Segments.....	12-24
Segment Space and the High Water Mark	12-27
Overview of Tablespaces	12-30
Permanent Tablespaces	12-31
Temporary Tablespaces.....	12-34
Tablespace Modes	12-34
Tablespace File Size	12-35

Part V Oracle Instance Architecture

13 Oracle Database Instance

Introduction to the Oracle Database Instance	13-1
Database Instance Structure	13-1
Database Instance Configurations	13-2
Overview of Instance Startup and Shutdown	13-5
Overview of Instance and Database Startup	13-5
Overview of Database and Instance Shutdown	13-8
Overview of Checkpoints	13-11
Purpose of Checkpoints	13-11
When Oracle Database Initiates Checkpoints.....	13-11
Overview of Instance Recovery	13-12
Purpose of Instance Recovery	13-12
When Oracle Database Performs Instance Recovery	13-12
Importance of Checkpoints for Instance Recovery	13-13
Instance Recovery Phases	13-14
Overview of Parameter Files	13-15

Initialization Parameters	13-15
Server Parameter Files	13-16
Text Initialization Parameter Files	13-16
Modification of Initialization Parameter Values	13-17
Overview of Diagnostic Files	13-18
Automatic Diagnostic Repository	13-19
Alert Log	13-21
Trace Files	13-22
14 Memory Architecture	
Introduction to Oracle Database Memory Structures	14-1
Basic Memory Structures	14-1
Oracle Database Memory Management	14-3
Overview of the User Global Area	14-3
Overview of the Program Global Area	14-4
Contents of the PGA	14-5
PGA Usage in Dedicated and Shared Server Modes	14-7
Overview of the System Global Area	14-8
Database Buffer Cache	14-9
Redo Log Buffer	14-14
Shared Pool	14-15
Large Pool	14-21
Java Pool	14-22
Streams Pool	14-23
Fixed SGA	14-23
Overview of Software Code Areas	14-23
15 Process Architecture	
Introduction to Processes	15-1
Multiple-Process Oracle Database Systems	15-1
Types of Processes	15-2
Overview of Client Processes	15-3
Client and Server Processes	15-4
Connections and Sessions	15-4
Overview of Server Processes	15-6
Dedicated Server Processes	15-6
Shared Server Processes	15-6
Overview of Background Processes	15-7
Mandatory Background Processes	15-7
Optional Background Processes	15-11
Slave Processes	15-13
16 Application and Networking Architecture	
Overview of Oracle Application Architecture	16-1
Overview of Client/Server Architecture	16-1
Overview of Multitier Architecture	16-3

Overview of Grid Architecture	16-5
Overview of Oracle Networking Architecture	16-5
How Oracle Net Services Works.....	16-6
The Oracle Net Listener.....	16-6
Dedicated Server Architecture	16-9
Shared Server Architecture.....	16-11
Database Resident Connection Pooling	16-14
Overview of the Program Interface.....	16-16
Program Interface Structure	16-16
Program Interface Drivers	16-16
Communications Software for the Operating System	16-16

Part VI Oracle Database Administration and Development

17 Topics for Database Administrators and Developers

Overview of Database Security	17-1
User Accounts.....	17-1
Authentication	17-3
Encryption	17-4
Access Control	17-4
Monitoring	17-5
Overview of High Availability	17-6
High Availability and Unplanned Downtime	17-6
High Availability and Planned Downtime	17-9
Overview of Grid Computing.....	17-11
Database Server Grid.....	17-12
Database Storage Grid.....	17-14
Overview of Data Warehousing and Business Intelligence	17-14
Data Warehousing and OLTP	17-15
Data Warehouse Architecture	17-16
Overview of Extraction, Transformation, and Loading (ETL)	17-18
Business Intelligence.....	17-19
Overview of Oracle Information Integration	17-20
Federated Access.....	17-20
Information Sharing.....	17-21

18 Concepts for Database Administrators

Duties of Database Administrators	18-1
Tools for Database Administrators	18-2
Oracle Enterprise Manager.....	18-2
SQL*Plus.....	18-4
Tools for Database Installation and Configuration.....	18-4
Tools for Oracle Net Configuration and Administration.....	18-4
Tools for Data Movement and Analysis	18-5
Topics for Database Administrators	18-9
Backup and Recovery	18-9

Memory Management	18-15
Resource Management and Task Scheduling	18-18
Performance Diagnostics and Tuning	18-20

19 Concepts for Database Developers

Duties of Database Developers	19-1
Tools for Database Developers	19-1
SQL Developer	19-2
Oracle Application Express	19-2
Oracle JDeveloper	19-2
Oracle JPublisher	19-3
Oracle Developer Tools for Visual Studio .NET	19-3
Topics for Database Developers	19-3
Principles of Application Design and Tuning	19-4
Client-Side Database Programming	19-5
Globalization Support	19-8
Unstructured Data	19-11

Glossary

Index

Preface

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

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 [Chapter 1, "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 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

This manual is intended to be read with the following manuals:

- *Oracle Database 2 Day DBA*
- *Oracle Database 2 Day Developer's Guide*

For more related documentation, see "[Oracle Database Documentation Roadmap](#)" on page 1-12.

Many manuals in the Oracle Database documentation set use the sample schemas of the seed database, which 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.
<i>italic</i>	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.

Introduction to Oracle Database

This chapter provides an overview of Oracle Database and contains the following sections:

- [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. 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. 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.

See Also: <http://portal.acm.org/citation.cfm?id=362685> for an abstract and link to Codd's paper

Relational Database Management System (RDBMS)

The relational model is the basis for a **relational database management system (RDBMS)**. Essentially, an RDBMS moves data into a database, stores the data, and retrieves it so that it can be manipulated by applications. 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 30 years of innovative development. Highlights in the evolution of Oracle Database include the following:

- Founding of Oracle

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 PCs. 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 enable 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.

See Also:

<http://www.oracle.com/technetwork/issue-archive/2007/07-jul/o4730-090772.html> for an article summarizing the evolution of Oracle Database

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 schema is owned by a database user and 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.

See Also: "Introduction to Schema Objects" on page 2-1

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 rules for each column of a table. These rules are called **integrity constraints**. 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" on page 2-6
- Chapter 5, "Data Integrity"

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: ["Overview of Indexes"](#) on page 3-1

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. In contrast to procedural languages such as C, which describe *how* things should be done, SQL is nonprocedural and describes *what* should be done. Users specify the result that they want (for example, the names of current employees), not how to derive it. 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. 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: [Chapter 7, "SQL"](#)

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 **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: [Chapter 8, "Server-Side Programming: PL/SQL and Java"](#) and ["Client-Side Database Programming"](#) on page 19-5

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

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

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 of the features that sets 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: [Chapter 10, "Transactions"](#)

Data Concurrency

A requirement of a multiuser RDBMS is the control of **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"](#) on page 9-11

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 dirty reads, which occur when one transaction sees uncommitted changes made by another concurrent transaction.

Oracle Database always enforces **statement-level read consistency**, which guarantees that the data returned by a single query is committed and consistent with respect to 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 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:

- [Chapter 9, "Data Concurrency and Consistency"](#)
- *Oracle Database Advanced Application Developer's Guide* to learn about 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**). Because an instance and a database are so closely connected, the term **Oracle database** is sometimes used to refer to both instance and database. In the strictest sense the terms have the following meanings:

- Database

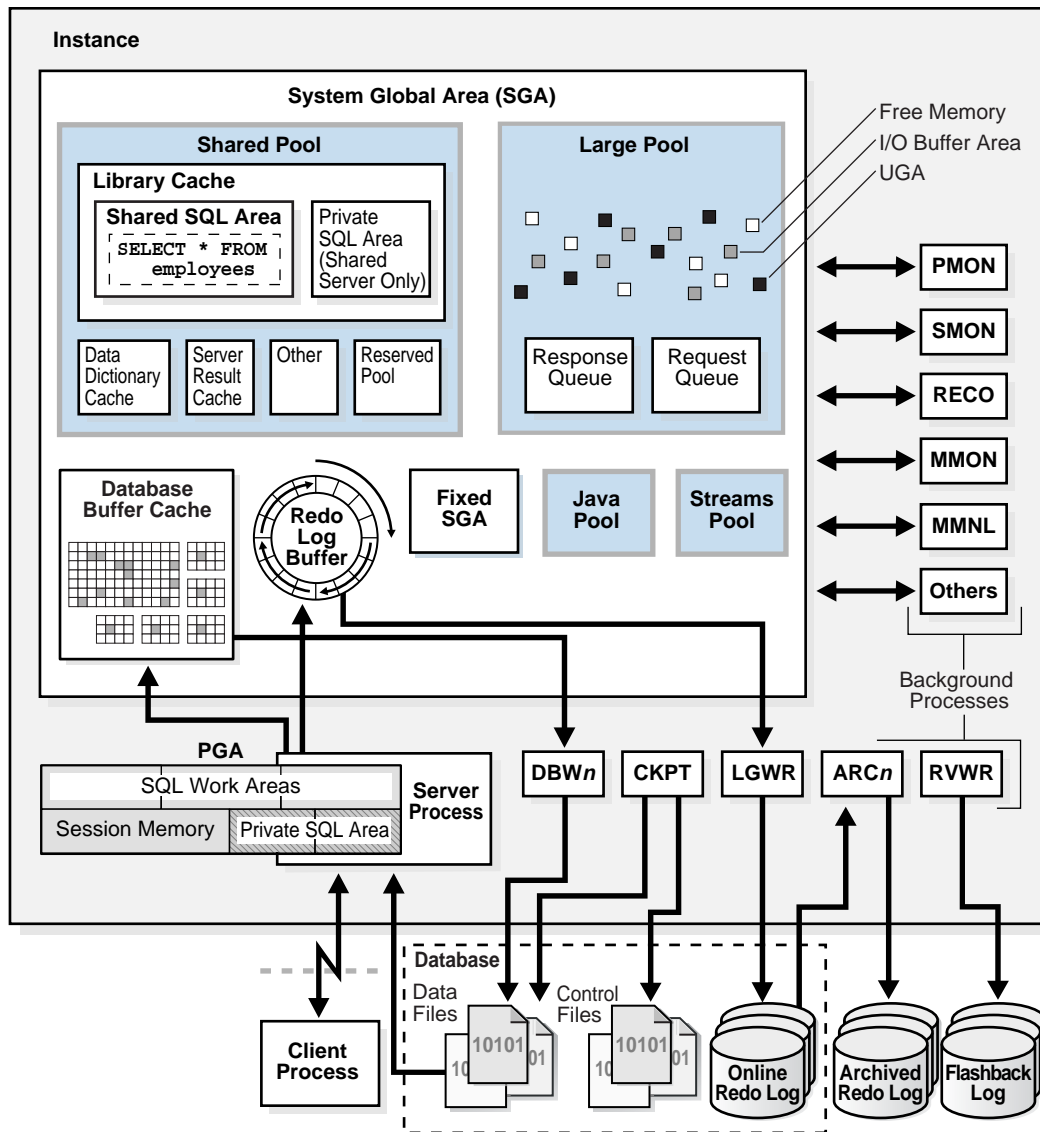
A database is a set of files, located on disk, that store data. These files can exist independently of a database instance.

- Database instance

An instance is a set of memory structures that manage database files. The 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.

[Figure 1–1](#) shows a database and its instance. For each user connection to the instance, the application is run by a **client process**. Each client process is associated with its own server process. The server process has its own private session memory, known as the **program global area (PGA)**.

Figure 1–1 Oracle Instance and Database



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, the physical storage of data can be managed 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.

See Also: [Chapter 13, "Oracle Database Instance"](#)

Database Storage Structures

An essential task of a relational database is data storage. This section briefly describes the physical and logical storage structures used by Oracle Database.

Physical Storage Structures

The physical database structures are the files that store the data. When you execute the SQL command `CREATE DATABASE`, the following files are created:

- Data files
Every Oracle database 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 Oracle database 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 Oracle Database 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 records**), which record all changes made to data.

Many other files are important for the functioning of an Oracle database server. These files include parameter files and diagnostic files. Backup files and **archived redo log files** are offline files important for backup and recovery.

See Also: [Chapter 11, "Physical Storage Structures"](#)

Logical Storage Structures

This section discusses logical storage structures. The following logical storage structures enable Oracle Database to have fine-grained control of disk space use:

- 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 a segment. Each tablespace contains at least one data file.

See Also: [Chapter 12, "Logical Storage Structures"](#)

Database Instance Structures

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

When applications connect to an Oracle database, they are connected 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*. For the purpose of this discussion, 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: [Chapter 15, "Process Architecture"](#)

Instance Memory Structures

Oracle Database creates and uses memory structures for purposes such as memory for program code, data shared among users, and private data areas for each connected user. The following memory structures are associated with an 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 cached data blocks and shared SQL areas.
- Program Global Areas (PGA)
A PGA is a memory region that contain 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: [Chapter 14, "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** refers to the computing environment in which a database application connects to an Oracle database. The two most common database architectures are client/server and multitier.

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.

In a traditional **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, thus lessening the load on the database. The application server can serve as an interface between clients and multiple databases and provide an additional level of security.

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.

See Also:

- ["Overview of Multitier Architecture"](#) on page 16-3
- *Oracle XML DB Developer's Guide* for more information about using Web services with the database

Networking 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 separate process that runs on the database server or elsewhere in the network. Client applications can send connection requests to the listener, which manages the traffic of these requests to the database server. 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 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 Networking Architecture"](#) on page 16-5
- *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

This section explains how this manual should be read and where it fits into the Oracle Database documentation set as a whole.

To a new user, the Oracle Database documentation library can seem daunting. Not only are there over 175 manuals, but many of these manuals are several hundred pages long. However, the documentation 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 (*Oracle Database 2 Day DBA*, *Oracle Database 2 Day Developer's Guide*, or this manual), 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 set.

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:

- *Oracle Database 2 Day DBA*

This manual is a task-based DBA quick start that teaches you how to perform day-to-day database administrative tasks. It teaches you how to perform all common administrative tasks needed to keep the database operational, including how to perform basic troubleshooting and performance monitoring activities.

- *Oracle Database 2 Day Developer's Guide*

This manual is a task-based database developer quick start guide that 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 references *Oracle Database Concepts* for conceptual background about a task.

Intermediate Group

The next step up from the basic group is the intermediate group. The manuals in this group are prefixed with the word *2 Day +* because they expand on and assume information contained in the *2 Day* manuals. These manuals cover topics in more depth than was possible in the basic manuals, or cover topics of special interest. As shown in [Table 1-1](#), the *2 Day +* manuals are divided into manuals for DBAs and developers.

Table 1–1 Intermediate Group: 2 Day + Guides

Database Administrators	Database Developers
<i>Oracle Database 2 Day + Performance Tuning Guide</i>	<i>Oracle Database 2 Day + Application Express Developer's Guide</i>
<i>Oracle Database 2 Day + Real Application Clusters Guide</i>	<i>Oracle Database 2 Day + Java Developer's Guide</i>
<i>Oracle Database 2 Day + Data Warehousing Guide</i>	<i>Oracle Database 2 Day + .NET Developer's Guide for Microsoft Windows</i>
<i>Oracle Database 2 Day + Data Replication and Integration Guide</i>	<i>Oracle Database 2 Day + PHP Developer's Guide</i>
<i>Oracle Database 2 Day + Security Guide</i>	

Advanced Group

The next step up from the intermediate group is the advanced group. These manuals are intended for expert users who require more detailed information about a particular topic than can be provided by the 2 Day + manuals. Essential reference manuals in the advanced group include:

- *Oracle Database SQL Language Reference*
This manual is the definitive source of information about Oracle SQL.
- *Oracle Database Reference*
The manual is the definitive source of information about initialization parameters, data dictionary views, and dynamic performance views.

The advanced guides are too numerous to list in this section. [Table 1–2](#) lists guides that are used by the majority of expert DBAs and developers at one time or another.

Table 1–2 Advanced Group

Database Administrators	Database Developers
<i>Oracle Database Administrator's Guide</i>	<i>Oracle Database Advanced Application Developer's Guide</i>
<i>Oracle Database Performance Tuning Guide</i>	<i>Oracle Database PL/SQL Language Reference</i>
<i>Oracle Database Backup and Recovery User's Guide</i>	<i>Oracle Database PL/SQL Packages and Types Reference</i>
<i>Oracle Real Application Clusters Administration and Deployment Guide</i>	

Other advanced guides required by a particular user depend on the area of responsibility of this user. For example, a security officer will naturally refer to the *Oracle Database Security Guide*.

Part I

Oracle Relational Data Structures

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

This part contains the following chapters:

- [Chapter 2, "Tables and Table Clusters"](#)
- [Chapter 3, "Indexes and Index-Organized Tables"](#)
- [Chapter 4, "Partitions, Views, and Other Schema Objects"](#)
- [Chapter 5, "Data Integrity"](#)
- [Chapter 6, "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.

This chapter contains the following sections:

- [Introduction to Schema Objects](#)
- [Overview of Tables](#)
- [Overview of Table Clusters](#)

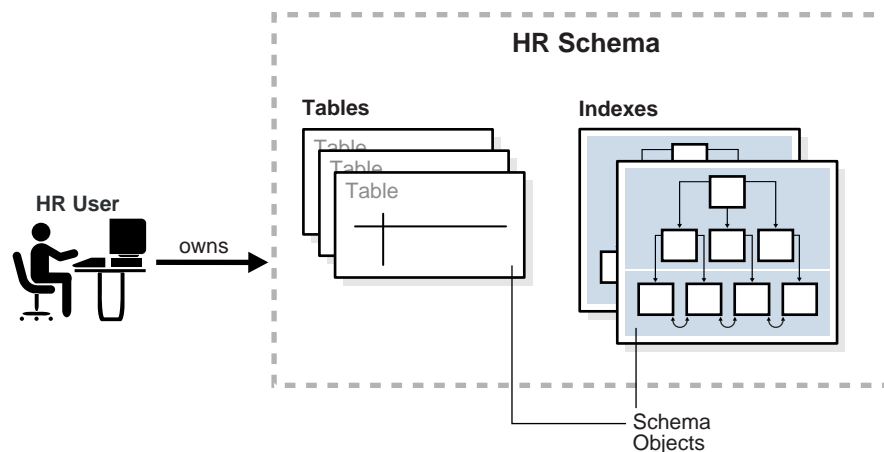
Introduction to Schema Objects

A database **schema** is a logical container for data structures, called **schema objects**. Examples of schema objects are tables and indexes. Schema objects are created and manipulated with **SQL**.

A **database user** has a password and various database **privileges**. Each user 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 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. [Figure 2-1](#) depicts a schema owner named `hr` and schema objects within the `hr` schema.

Figure 2-1 HR Schema



See Also: ["Overview of Database Security"](#) on page 17-1 to learn more about users and privileges

Schema Object Types

The most important schema objects in a relational database are tables. A **table** stores data in rows.

Oracle SQL enables you to create and manipulate many other types of schema objects, including the following:

- **Indexes**

Indexes are schema objects that contains 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. See [Chapter 3, "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. See ["Overview of Partitions"](#) on page 4-1.
 - **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. See ["Overview of Views"](#) on page 4-12.
 - **Sequences**

A sequence is a user-created object that can be shared by multiple users to generate integers. Typically, sequences are used to generate **primary key** values. See ["Overview of Sequences"](#) on page 4-20.
 - **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. See ["Overview of Dimensions"](#) on page 2-22.
 - **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**. See ["Overview of Synonyms"](#) on page 4-22.
 - **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. See ["PL/SQL Subprograms"](#) on page 8-3 and ["PL/SQL Packages"](#) on page 8-6.
- 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 users, **roles**, **contexts**, and **directory objects**.

See Also:

- *Oracle Database 2 Day DBA and 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

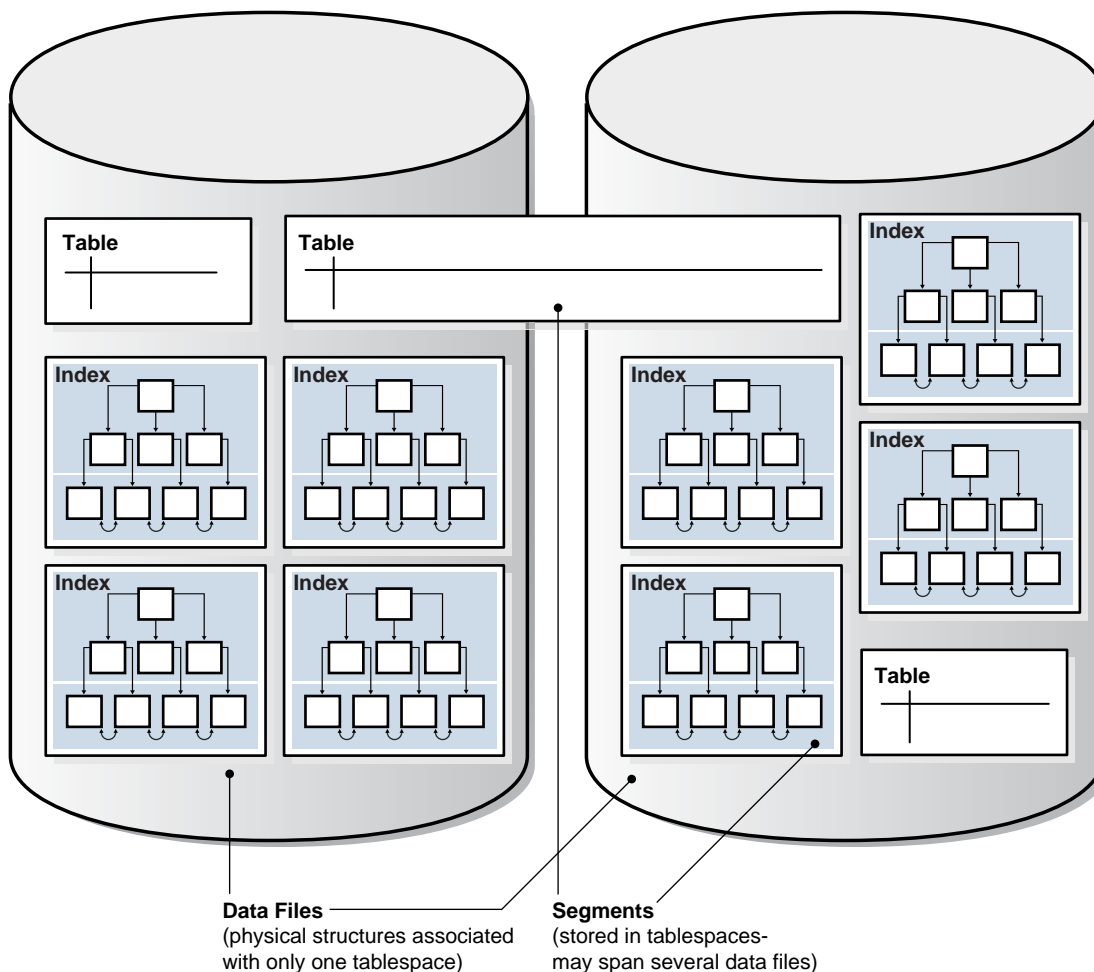
Schema Object Storage

Some schema objects store data in logical storage structures called **segments**. 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 section 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.

Figure 2–2 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 2–2 Segments, Tablespaces, and Data Files



See Also:

- [Chapter 12, "Logical Storage Structures"](#) to learn about tablespaces and segments
- *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn how to manage storage for schema objects

Schema Object Dependencies

Some schema objects reference other objects, creating **schema object dependencies**. For example, a view contains a **query** that references tables or other views, while a **PL/SQL** subprogram invokes other subprograms. If the definition of object A references object B, then A is a **dependent object** with respect to B and B is a **referenced object** with respect to 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 a dependent object is created, 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 dependent object is marked 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 in, 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 Advanced Application Developer's Guide* to learn how to manage schema object dependencies

SYS and SYSTEM Schemas

All Oracle databases include default administrative accounts. 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 administrative account `SYS` 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` 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:

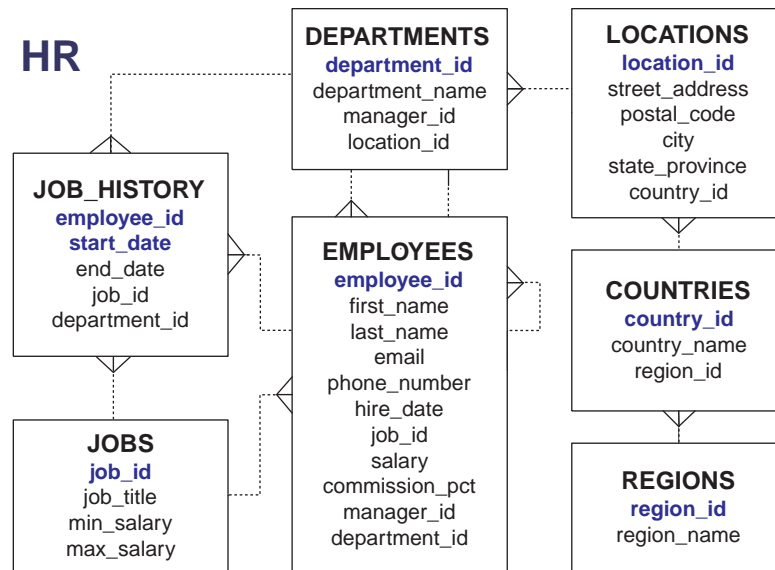
- ["User Accounts"](#) on page 17-1 and ["Connection with Administrator Privileges"](#) on page 13-6
- *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn about `SYS`, `SYSTEM`, and other administrative accounts

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` schema is a sample schema that contains information about employees, departments and locations, work histories, and so on.

Figure 2–3 is an entity-relationship diagram of the tables in the `hr` schema. Most examples in this manual use objects from this schema.

Figure 2–3 HR Schema



See Also: *Oracle Database 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 2–1](#) on page 2-8 shows a `CREATE TABLE` statement for a relational table.
- **Object tables**
The columns correspond to the top-level attributes of an **object type**. See "Object Tables" on page 2-15.

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" on page 3-20.

- An **external table** is a read-only table whose metadata is stored in the database but whose data is stored outside the database. See "[External Tables](#)" on page 2-16.

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 section contains the following topics:

- [Columns and Rows](#)
- [Example: CREATE TABLE and ALTER TABLE Statements](#)
- [Oracle Data Types](#)
- [Integrity Constraints](#)
- [Object Tables](#)
- [Temporary Tables](#)
- [External Tables](#)
- [Table Storage](#)
- [Table Compression](#)

See Also: *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn how to manage tables

Columns and Rows

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`.

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.

After you create a table, you can insert, query, delete, and update rows using SQL. 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.

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

Example: CREATE TABLE and ALTER TABLE Statements

The Oracle SQL command to create a table is `CREATE TABLE`. [Example 2-1](#) 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.

Example 2–1 CREATE TABLE employees

```

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 2–2](#) 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.

Example 2–2 ALTER TABLE employees

```

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 2–3](#) shows 8 rows and 6 columns of the hr.employees table.

Example 2–3 Rows in the 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 output in [Example 2–3](#) 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 104 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.

See Also:

- ["Overview of LOBs"](#) on page 19-12
- *Oracle Database SQL Language Reference* to learn about built-in SQL data types
- *Oracle Database PL/SQL Language Reference* to learn about PL/SQL data types
- *Oracle Database Advanced Application Developer's Guide* for information about how to use the built-in data types

Character Data Types

Character data types store character (alphanumeric) data in strings. The most commonly used 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** or **code page**. 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 can be measured in bytes or characters. **Byte semantics** treat strings as a sequence of bytes. This is the default for character data types. **Character semantics** treat strings as a sequence of characters. A character is technically a code point of the database character set.

See Also:

- ["Character Sets"](#) on page 19-9
- *Oracle Database 2 Day Developer's Guide* and *Oracle Database Advanced Application Developer's Guide* and to learn how to select a character data type

VARCHAR2 and CHAR Data Types The `VARCHAR2` data type stores variable-length character literals. The terms **literal** and **constant value** are synonymous and refer to 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 2-1](#), the `last_name` column has a data type of `VARCHAR2(25)`, which means that any name stored in the column can have 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.

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 2-1](#), 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, while `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. The `hire_date` column in [Example 2-1](#) has a `DATE` data type.

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. See *Oracle Database Advanced Application Developer's Guide*.

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-09`.

`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-09`. If the date format uses `RR`, then `09` specifies 2009, whereas if the format uses `YY`, then `09` specifies 1909. You can change the default date format at both the **instance** and the **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 Advanced Application Developer's Guide* for more information about centuries and date format masks
- *Oracle Database SQL Language Reference* for information about datetime format codes

TIMESTAMP Data Type The `TIMESTAMP` data type is an extension of the `DATE` data type. It 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. 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 kinds of rowids.

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 the fastest means of accessing particular rows.
- Rowids provide the ability to see how a table is organized.
- Rowids are unique identifiers for rows in a given table.

You can also create tables with columns defined using the ROWID data type. For example, you can define an exception table with a column of data type ROWID to store the rowids of rows that violate integrity constraints. Columns defined using the ROWID data type behave like other table columns: values can be updated, and so on.

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.

[Example 2-4](#) queries the ROWID pseudocolumn to show the rowid of the row in the `employees` table for employee 100.

Example 2-4 ROWID Pseudocolumn

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

```
ROWID
```

```
-----
AAAPecAAFAAAAABSAAA
```

See Also:

- ["Rowid Format"](#) on page 12-10
- *Oracle Database Advanced Application Developer's 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')
       2 FROM employees
       3 WHERE department_id = 80 AND last_name = 'Russell';
```

EMPLOYEE	TO_CHAR(SAL
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

Integrity constraints are named rules that restrict the values for one or more columns in a table. These 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. Data that does not conform to the constraint is prevented from being entered. If a constraint is disabled, then data that does not conform to the constraint can be allowed to enter the database.

In [Example 2-1](#) on page 2-8, 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. Constraints can be temporarily disabled if needed. The database stores constraints in the [data dictionary](#).

See Also:

- [Chapter 5, "Data Integrity"](#) to learn about integrity constraints
- *Oracle Database SQL Language Reference* to learn about SQL constraint clauses

Object Tables

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. [Example 2–5](#) creates an object type named `department_typ`.

Example 2–5 Object Type

```
CREATE TYPE department_typ AS OBJECT
  ( d_name    VARCHAR2(100),
    d_address VARCHAR2(200) );
/
```

An **object table** is a special kind of table in which each row represents an object. The `CREATE TABLE` statement in [Example 2–6](#) creates an object table named `departments_obj_t` of the object type `department_typ`. The attributes (columns) of this table are derived from the definition of the object type. The `INSERT` statement inserts a row into this table.

Example 2–6 Object Table

```
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

Temporary Tables

Oracle Database **temporary tables** hold data that exists only for the duration of a transaction or session. Data in a temporary table is private to the session, which means that each session can only see and modify its own data.

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. Each schedule is represented by a row in a temporary table. During the session, the schedule data is private. When the student decides on a schedule, the application moves the row for the chosen schedule to a permanent table. At the end of the session, the schedule data in the temporary data is automatically dropped.

Temporary Table Creation

The `CREATE GLOBAL TEMPORARY TABLE` statement creates a temporary table. The `ON COMMIT` clause specifies whether the table data is transaction-specific (default) or session-specific.

Unlike temporary tables in some other relational databases, when you create a temporary table in an Oracle database, you create a static table definition. The temporary table is a persistent object described in the data dictionary, but appears empty until your session inserts data into the table. You create a temporary table for the database itself, not for every PL/SQL [stored procedure](#).

Because temporary tables are statically defined, you can create indexes for them with the `CREATE INDEX` statement. Indexes created on temporary tables 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 temporary table.

See Also:

- *Oracle Database Administrator's Guide* to learn how create and manage temporary tables
- *Oracle Database SQL Language Reference* for `CREATE GLOBAL TEMPORARY TABLE` syntax and semantics
- ["Overview of Views"](#) on page 4-12 and ["Overview of Triggers"](#) on page 8-16

Segment Allocation in Temporary Tables

Like permanent tables, temporary tables are defined in the data dictionary. Temporary segments are allocated when data is first inserted. Until data is loaded in a session the table appears empty. Temporary segments are deallocated at the end of the transaction for transaction-specific temporary tables and at the end of the session for session-specific temporary tables.

See Also: ["Temporary Segments"](#) on page 12-23

External Tables

An **external table** accesses data in external sources as if this data were in a table in the database. You can use SQL, PL/SQL, and Java to query the external data.

External tables are useful for querying flat files. For example, a SQL-based application may need to access records in a text file. The 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 file to the location specified in the external table definition, and use SQL to query the records in the text file.

External tables are also valuable for performing [ETL](#) tasks common in [data warehouse](#) environments. For example, external tables enable the pipelining of the data loading phase with the transformation phase, eliminating the need to stage data inside the

database in preparation for further processing inside the database. See "[Overview of Data Warehousing and Business Intelligence](#)" on page 17-14.

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.

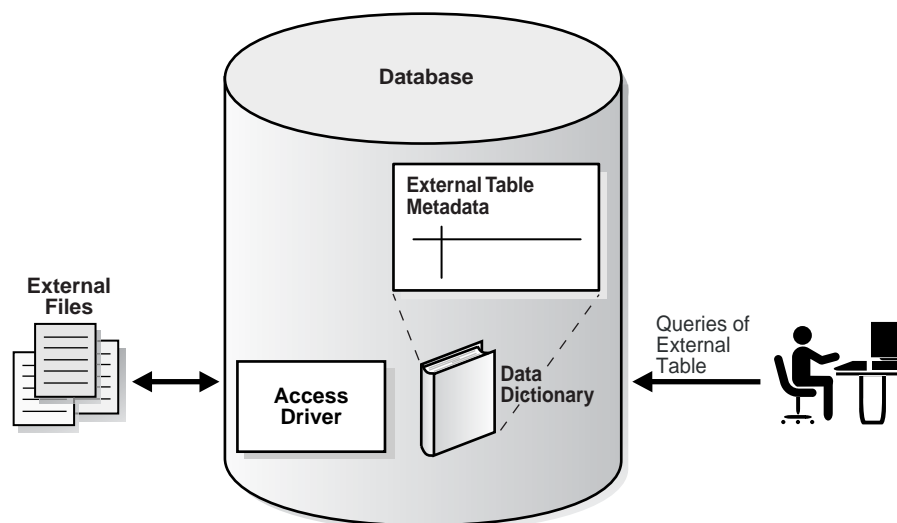
A `CREATE TABLE ... ORGANIZATION EXTERNAL` statement has two parts. The **external table definition** describes the column types. This definition is like a view that enables SQL to query external data without loading it into the database. The second part of the statement maps the external data to the 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, virtual columns, and column objects.

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. [Figure 2-4](#) represents how external data is accessed.

Figure 2-4 External Tables



Oracle provides the `ORACLE_LOADER` (default) and `ORACLE_DATAPUMP` access drivers for external tables. For both drivers, the external files are not Oracle data files.

`ORACLE_LOADER` enables read-only access to external files using `SQL*Loader`. You cannot create, update, or append to an external file using the `ORACLE_LOADER` driver.

The `ORACLE_DATAPUMP` driver enables you to **unload** external data. This operation involves reading data from the database and inserting 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. The driver also enables you to **load** external data, which involves reading an external table and loading its data into a database.

See Also:

- *Oracle Database Administrator's Guide* to learn about managing external tables, external connections, and [directory objects](#)
- *Oracle Database Utilities* to learn about external tables
- *Oracle Database SQL Language Reference* for information about creating and querying external tables

Table Storage

Oracle Database uses a [data segment](#) in a tablespace to hold table data. As explained in "[User Segments](#)" on page 12-21, a segment contains [extents](#) made up of [data blocks](#).

The data segment for a table (or cluster data segment, when dealing with 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. 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.

Note: Index-organized tables use a different principle of organization. See "[Overview of Index-Organized Tables](#)" on page 3-20.

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 [Example 2-7](#).

Example 2-7 Rows in Departments Table

```
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: *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 (see "[Data Block Format](#)" on page 12-7).

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.

Rowids of Row Pieces

A **rowid** is effectively a 10-byte physical address of a row. As explained in "[Rowid Data Types](#)" on page 2-13, 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 (see "[B-Tree Indexes](#)" on page 3-5). 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](#)" on page 12-10

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 and OLTP Table 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 loads, `ALTER TABLE . . . MOVE` operations, or online table redefinition to achieve basic compression.

- **OLTP table compression**

This type of compression is intended for OLTP applications and compresses data manipulated by any SQL operation.

For basic and OLTP table 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 (see [Figure 12-7](#) on page 12-9). Duplicate values are replaced with a short reference to a symbol table stored at the beginning of the block. Thus, information needed 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.

The following statement applies OLTP compression to the `orders` table:

```
ALTER TABLE oe.orders COMPRESS FOR OLTP;
```

The following example of a partial `CREATE TABLE` statement specifies OLTP compression for one partition and basic 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_2008 VALUES LESS THAN(TO_DATE(...)) COMPRESS BASIC,
  partition sales_2009 VALUES LESS THAN (MAXVALUE) COMPRESS FOR OLTP );
```

See Also:

- ["Data Block Compression"](#) on page 12-11 to learn about the format of compressed data blocks
- *Oracle Database Administrator's Guide* and *Oracle Database Performance Tuning Guide* to learn about table compression
- ["SQL*Loader"](#) on page 18-5 to learn about using SQL*Loader for direct path loads

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.

Types of Hybrid Columnar Compression If your underlying storage supports Hybrid Columnar Compression, then you can specify the following compression types, depending on your requirements:

- Warehouse compression

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

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

To achieve warehouse or online archival compression, you must use direct path loads, ALTER TABLE . . . MOVE operations, or online table redefinition.

Hybrid Columnar Compression is optimized for Data Warehousing and decision support applications on Exadata storage. 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 Exadata storage server.

Other Oracle storage systems support Hybrid Columnar Compression, and deliver the same space savings as on 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. [Table 2–1](#) shows a subset of the rows in `daily_sales`.

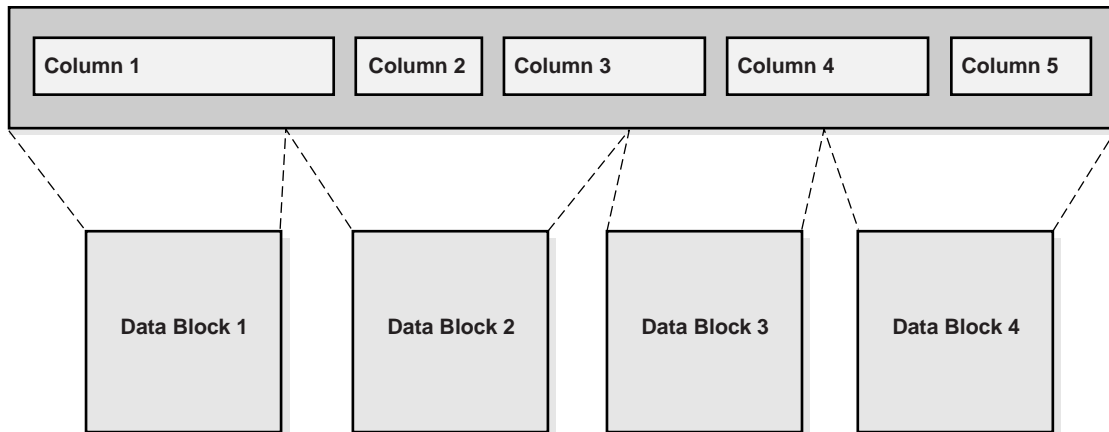
Table 2–1 Sample Table `daily_sales`

Item_ID	Date	Num_Sold	Shipped_From	Restock
1000	01-JUN-11	2	WAREHOUSE1	Y
1001	01-JUN-11	0	WAREHOUSE3	N
1002	01-JUN-11	1	WAREHOUSE3	N
1003	01-JUN-11	0	WAREHOUSE2	N
1004	01-JUN-11	2	WAREHOUSE1	N
1005	01-JUN-11	1	WAREHOUSE2	N

Assume that the rows in [Table 2–1](#) are 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 [Figure 2–5](#), each compression unit can span multiple data blocks. The values for a particular column may or may not span multiple blocks.

Figure 2–5 Compression Unit



Hybrid Columnar Compression has implications for row locking (see "[Row Locks \(TX\)](#)" on page 9-18). When an update occurs for a row in an uncompressed data block, only the updated row is locked. In contrast, the database must lock all rows in the compression unit if an update is made to any row in the unit. Updates to rows using Hybrid Columnar Compression cause rowids to change.

Note: When tables use Hybrid Columnar Compression, Oracle DML locks larger blocks of data (compression units), which may reduce concurrency.

See Also:

- *Oracle Database Licensing Information* to learn about licensing requirements for Hybrid Columnar Compression
- *Oracle Database Administrator's Guide* to learn how to use Hybrid Columnar Compression

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 employees folders and the other with boxes of departments 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 a different box, and so on.

You can 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.

See Also: *Oracle Database Performance Tuning Guide* for guidelines on when to use table clusters

Overview of Indexed Clusters

An **indexed 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.

Assume that you create the cluster `employees_departments_cluster` with the cluster key `department_id`, as shown in [Example 2–8](#). Because the `HASHKEYS` clause is not specified, this cluster is an indexed cluster. Afterward, you create an index named `idx_emp_dept_cluster` on this cluster key.

Example 2–8 Indexed Cluster

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

```
CREATE INDEX idx_emp_dept_cluster ON CLUSTER employees_departments_cluster;
```

You then 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):

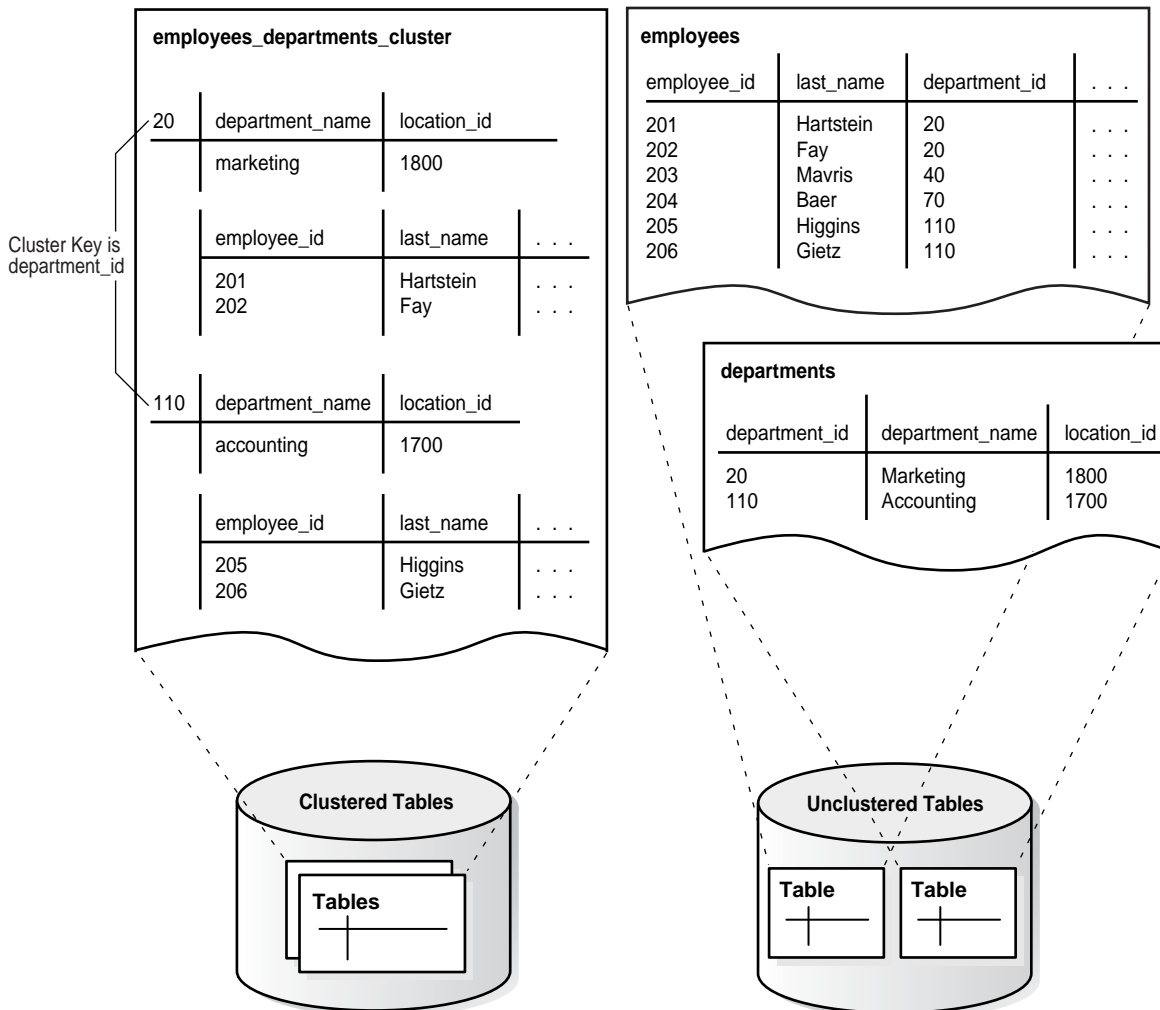
```
CREATE TABLE employees ( ... )
  CLUSTER employees_departments_cluster (department_id);

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

Finally, 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 2–6 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 2–6 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:

- ["Overview of Indexes"](#) on page 3-1
- *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

The **cluster key**, like the key of an indexed cluster, is a single column or composite key shared by the tables in the cluster. The **hash key values** are actual or possible values 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).

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. In [Example 2-9](#), the number of departments that are likely to exist is 100, so `HASHKEYS` is set to 100.

Example 2-9 Hash Cluster

```
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 described in [Example 2-8](#).

See Also: *Oracle Database Administrator's Guide* to learn how to create and manage hash clusters

Hash Cluster Queries

The database, not the user, determines how to hash the key values input by the user. For example, assume that users frequently execute queries such as the following, entering different department ID numbers for `p_id`:

```

SELECT *
FROM   employees
WHERE  department_id = :p_id;

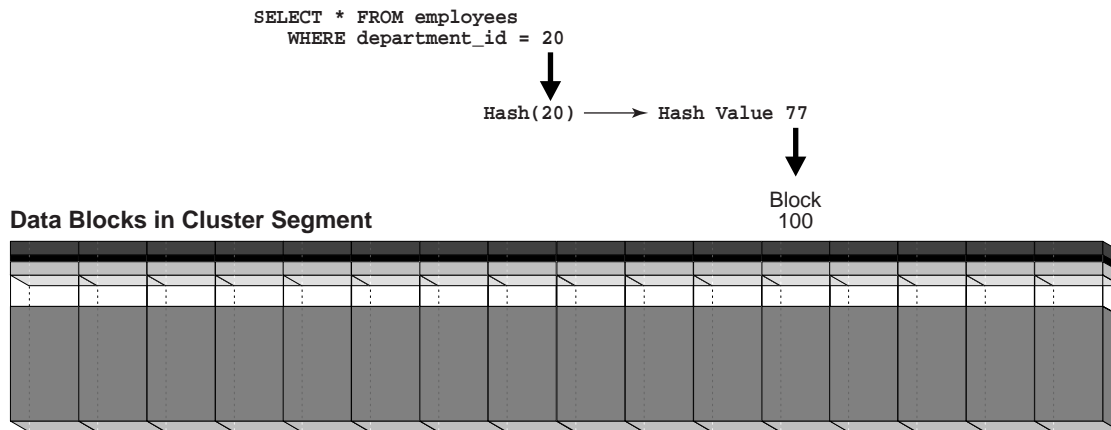
SELECT *
FROM   departments
WHERE  department_id = :p_id;

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.

Figure 2-7 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 2-7 Retrieving Data from a Hash Cluster



A limitation of hash clusters is the unavailability of range scans on nonindexed cluster keys (see "Index Range Scan" on page 3-7). Assume that no separate index exists for the hash cluster created in Example 2-9. 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.

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 [Example 2-9](#), `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:

$$\text{HASHKEYS} * \text{SIZE} / \text{database_block_size}$$

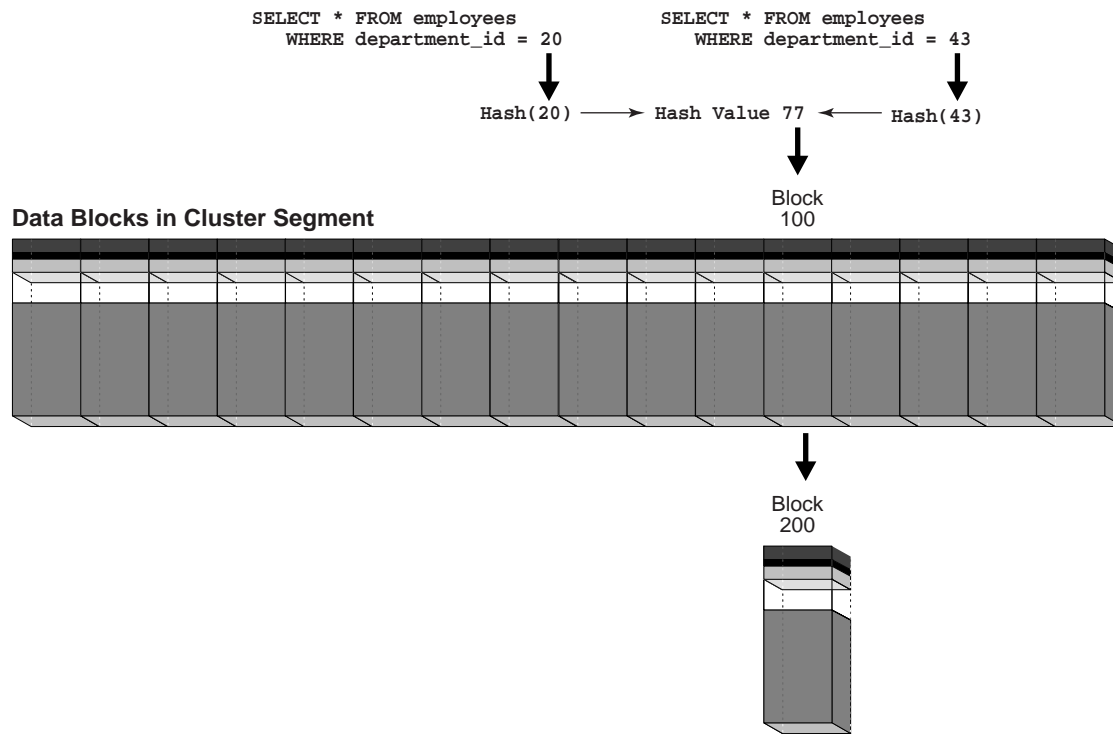
Thus, if the block size is 4096 bytes in [Example 2-9](#), then the database allocates at least 200 blocks to the hash cluster.

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 2-7](#) 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 2-8](#), 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 2–8 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

Indexes and Index-Organized Tables

This chapter discusses indexes, which are schema objects that can speed access to table rows, and index-organized tables, which are tables stored in an index structure.

This chapter contains the following sections:

- [Overview of Indexes](#)
- [Overview of Index-Organized Tables](#)

Overview of Indexes

An **index** is an optional structure, associated with a table or **table cluster**, that can sometimes speed data access. By creating an **index** on one or more columns of a table, you gain the ability in some cases to 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, without an index, a **query** of location 2700 in the `hr.departments` table requires the database to search every row in every table block for this value. This approach does not scale well as data volumes increase.

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.

In general, consider creating an index on a column in any of 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: [Chapter 5, "Data Integrity"](#)

Index Characteristics

Indexes are schema objects that are logically and physically independent of the data in the objects with which they are associated. Thus, an index can be dropped or created without physically affecting the table for the index.

Note: If you drop an index, then applications still work. However, access of previously indexed data can be slower.

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.

The database automatically maintains and uses indexes after they are created. The database also automatically reflects changes to data, such as adding, updating, and deleting rows, in all relevant indexes with no additional actions required by users. 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.

Indexes have the following properties:

- Usability

Indexes are usable (default) or unusable. An **unusable index** is not maintained by DML operations and is ignored by the **optimizer**. 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

Indexes are visible (default) or invisible. An **invisible index** is maintained by DML operations and 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"](#) on page 7-10
- *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn how to manage indexes
- *Oracle Database Performance Tuning Guide* to learn how to tune indexes

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: *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. Columns in a composite index should appear in the order that makes the most sense for the queries that will retrieve data and 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](#)" on page 3-8).

Multiple indexes can exist for the same table if the permutation of columns differs for each index. You can create multiple indexes using the same columns if you specify distinctly different permutations of the columns. For example, the following SQL statements specify valid 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 Performance Tuning Guide* for more information about using composite indexes

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 column. For example, no two employees can have the same employee ID. Thus, 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 duplicate 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. The indexes can be categorized as follows:

- B-tree indexes
 - These indexes are the standard index type. They are excellent for primary key and highly-selective indexes. Used as concatenated indexes, B-tree indexes can retrieve data sorted by the indexed columns. B-tree indexes have the following subtypes:
 - Index-organized tables
 - An index-organized table differs from a heap-organized because the data is itself the index. See "[Overview of Index-Organized Tables](#)" on page 3-20.
 - 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. See "[Reverse Key Indexes](#)" on page 3-11.
 - Descending indexes
 - This type of index stores data on a particular column or columns in descending order. See "[Ascending and Descending Indexes](#)" on page 3-11.
 - B-tree cluster indexes
 - This type of index is used to index a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key. See "[Overview of Indexed Clusters](#)" on page 2-23.
- 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. See "[Bitmap Indexes](#)" on page 3-13.
- 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. See "[Function-Based Indexes](#)" on page 3-17.
- Application domain indexes

This type of index is created by a user 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. See "[Application Domain Indexes](#)" on page 3-19.

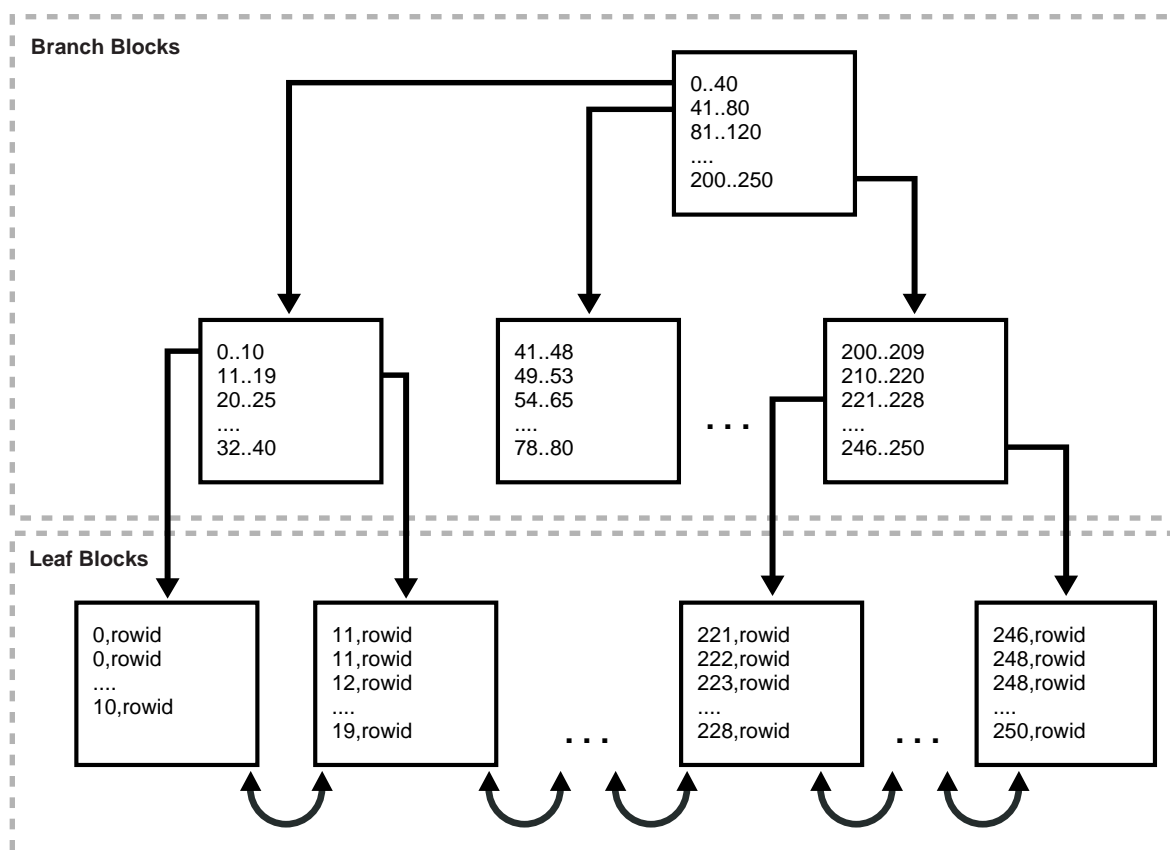
See Also: *Oracle Database Performance Tuning Guide* to learn about different index types

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.

Figure 3-1 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 3-1 Internal Structure of a B-tree Index



Branch Blocks and Leaf Blocks

A B-tree index has two types of blocks: **branch blocks** for searching and **leaf blocks** that store values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks. In Figure 3-1, 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 3-1](#), 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 3-1](#) 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 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.

See Also: *Oracle Database Performance 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.

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;
```

Also assume that 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.

For example, 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 it that prevents nulls from being considered in the query result set

For example, 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
Zlotkey,10500,rowid
Austin,7200,rowid
Baer,10000,rowid
Atkinson,2800,rowid
Austin,4800,rowid
.
.
.
```

Index Range Scan An **index range scan** is 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.

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 a `UNIQUE` index key 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. For example, assume that you run the following query for a customer in the `sh.customers` table:

```
SELECT * FROM sh.customers WHERE cust_email = 'Abbey@company.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`). [Example 3-1](#) shows a portion of the index entries.

Example 3-1 Composite Index Entries

```
F,Wolf@company.com,rowid
F,Wolsey@company.com,rowid
F,Wood@company.com,rowid
F,Woodman@company.com,rowid
F,Yang@company.com,rowid
F,Zimmerman@company.com,rowid
M,Abbassi@company.com,rowid
M,Abbey@company.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 [Example 3-1](#), 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.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.com'
UNION ALL
SELECT * FROM sh.customers WHERE cust_gender = 'M'
      AND cust_email = 'Abbey@company.com';
```

See Also: *Oracle Database Performance 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. The more order that exists in row storage for this value, the lower the clustering factor.

The clustering factor is useful as a rough measure of the number of I/Os required to read an entire table by means of 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

For example, assume that the employees table fits into two data blocks. [Table 3–1](#) depicts the rows in the two data blocks (the ellipses indicate data that is not shown).

Table 3–1 Contents of Two Data Blocks in the Employees Table

Data Block 1					Data Block 2				
100	Steven	King	SKING	...					
156	Janette	King	JKING	...					
115	Alexander	Khoo	AKHOO	...					
.									
.					149	Eleni	Zlotkey	EZLOTKEY	...
.					200	Jennifer	Whalen	JWHALEN	...
116	Shelli	Baida	SBaida				
204	Hermann	Baer	HBAER				
105	David	Austin	DAUSTIN				
130	Mozhe	Atkinson	MATKINSO	...	137	Renske	Ladwig	RLADWIG	...
166	Sundar	Ande	SANDE	...	173	Sundita	Kumar	SKUMAR	...
174	Ellen	Abel	EABEL	...	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
.
.
.
```

[Example 3–2](#) queries the ALL_INDEXES view for the clustering factor for these two indexes. 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.

Example 3–2 Clustering Factor

```
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

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.

See Also: *Oracle Database Performance Tuning Guide* to learn about design considerations for reverse key indexes

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 [Figure 3-1](#) 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 Performance 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

Key Compression

Oracle Database can use **key compression** to compress portions of the primary key column values in a B-tree index or an index-organized table. Key compression can greatly reduce the space consumed by the index.

In general, index keys have two pieces, a **grouping piece** and a **unique piece**. Key compression breaks the index key into 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. For example, suppose that you create a composite index on the `oe.orders` table as follows:

```
CREATE INDEX orders_mod_stat_ix ON orders ( order_mode, order_status );
```

Many repeated values occur in the `order_mode` and `order_status` columns. An index block may have entries as shown in [Example 3-3](#).

Example 3-3 Index Entries in Orders Table

```
online,0,AAAPvCAAFAAAAFaAAa
online,0,AAAPvCAAFAAAAFaAAg
online,0,AAAPvCAAFAAAAFaAAL
online,2,AAAPvCAAFAAAAFaAAm
online,3,AAAPvCAAFAAAAFaAAq
online,3,AAAPvCAAFAAAAFaAAt
```

In [Example 3-3](#), the key prefix would consist of a concatenation of the `order_mode` and `order_status` values. If this index were created with default key compression, then duplicate key prefixes such as `online,0` and `online,2` would be compressed. Conceptually, the database achieves compression as shown in the following example:

```
online,0
AAAPvCAAFAAAAFaAAa
AAAPvCAAFAAAAFaAAg
AAAPvCAAFAAAAFaAAL
```



```

online,2
AAPvCAAFAAAAFaAAm
online,3
AAPvCAAFAAAAFaAAq
AAPvCAAFAAAAFaAAt

```

Suffix entries 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 entry.

Alternatively, you could specify a prefix length when creating a compressed index. For example, if you specified prefix length 1, then the prefix would be `order_mode` and the suffix would be `order_status,rowid`. For the values in [Example 3-3](#), the index would factor out duplicate occurrences of `online` as follows:

```

online
0,AAPvCAAFAAAAFaAAa
0,AAPvCAAFAAAAFaAAg
0,AAPvCAAFAAAAFaAA1
2,AAPvCAAFAAAAFaAAm
3,AAPvCAAFAAAAFaAAq
3,AAPvCAAFAAAAFaAAt

```

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 key compression for partitioned indexes
- *Oracle Database SQL Language Reference* for descriptions of the `key_compression` clause of `CREATE INDEX`

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.customer` 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 `customer.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.

See Also:

- *Oracle Database Performance Tuning Guide* to learn how to use bitmap indexes for performance
- *Oracle Database Data Warehousing Guide* to learn how to use bitmap indexes in a data warehouse

Bitmap Indexes on a Single Table

[Example 3-4](#) shows a query of the sh.customers table. Some columns in this table are candidates for a bitmap index.

Example 3-4 Query of customers Table

```
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 3-2](#) illustrates the bitmap index for the cust_gender column output shown in [Example 3-4](#). It consists of two separate bitmaps, one for each gender.

Table 3-2 Sample Bitmap

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 its 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 3-3](#). To identify the customers who satisfy the criteria, Oracle Database can use the resulting bitmap to access the table.

Table 3-3 Sample Bitmap

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
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:

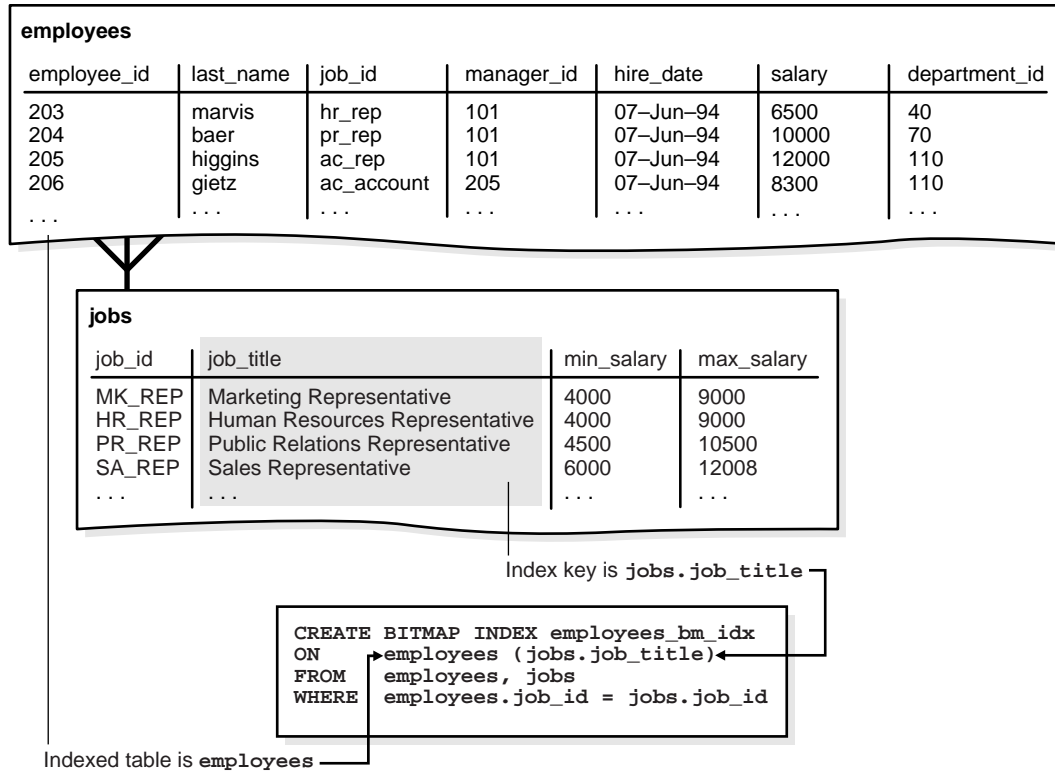
```
SELECT COUNT(*)
FROM   employees, jobs
WHERE  employees.job_id = jobs.job_id
AND    jobs.job_title = 'Accountant';
```

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 Figure 3–2, the index key is `jobs.job_title` and the indexed table is `employees`.

Figure 3–2 Bitmap Join Index



Conceptually, `employees_bm_idx` is an index of the `jobs.title` column in the SQL query shown in Example 3–5 (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.

Example 3–5 Join of employees and jobs Tables

```

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	AAAQNKAFFAAAABSAAAL
Accountant	AAAQNKAFFAAAABSAAAN
Accountant	AAAQNKAFFAAAABSAAAM
Accountant	AAAQNKAFFAAAABSAAAJ
Accountant	AAAQNKAFFAAAABSAAK
Accounting Manager	AAAQNKAFFAAAABTAAH
Administration Assistant	AAAQNKAFFAAAABTAAC
Administration Vice President	AAAQNKAFFAAAABSAAAC
Administration Vice President	AAAQNKAFFAAAABSAAAB
.	.

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 the index data is stored in one B-tree. The individual bitmaps are stored in the leaf blocks.

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,AAAPzRAAFAAAABSAAAd,AAAPzRAAFAAAABSAAt,0101001001
Stock Clerk,AAAPzRAAFAAAABSAAu,AAAPzRAAFAAAABSABz,100001
.
.
.
```

The same job title appears in multiple entries because the rowid range differs.

Assume that 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" on page 12-21

Function-Based Indexes

You can create indexes on functions and expressions that involve one or more columns in the table being indexed. A **function-based index** computes the value of a function or expression involving one or more columns and stores it in the index. A function-based index can be either a B-tree or a bitmap index.

The function used for building the index 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.

See Also:

- *Oracle Database Administrator's Guide* to learn how to create function-based indexes
- *Oracle Database Performance Tuning 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.

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 [Example 3–6](#) (partial sample output included).

Example 3–6 Query Containing an Arithmetic Expression

```
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

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';
```

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

The **optimizer** can use an index range scan on a function-based index for queries with expressions in `WHERE` clause. The range scan **access path** is especially beneficial when the predicate (`WHERE` clause) has low **selectivity**. In [Example 3-6](#) the optimizer can use an index range scan if an index is built on the expression `12*salary*commission_pct`.

A **virtual column** is 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.

See Also:

- "[Overview of the Optimizer](#)" on page 7-10
- *Oracle Database Performance Tuning Guide* for more information about gathering statistics
- *Oracle Database Administrator's Guide* to learn how to add virtual columns to a table

Application Domain Indexes

An **application domain index** is a customized index specific to an application. Oracle Database provides **extensible indexing** to do the following:

- Accommodate indexes on customized, complex data types such as documents, spatial data, images, and video clips (see "[Unstructured Data](#)" on page 19-11)
- Make use of specialized indexing techniques

You can encapsulate application-specific index management routines as an **indextype** schema object and 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 Data Cartridge Developer's Guide* for information about using data cartridges within the Oracle Database extensibility architecture

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: [Chapter 12, "Logical Storage Structures"](#)

Overview of Index-Organized Tables

An **index-organized table** is a table stored in a variation of a B-tree index structure. In a **heap-organized table**, rows are inserted 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. This type of table is often used for information retrieval, spatial (see "[Overview of Oracle Spatial](#)" on page 19-14), and **OLAP** applications (see "[OLAP](#)" on page 17-19).

See Also:

- *Oracle Database Administrator's Guide* to learn how to manage index-organized tables
- *Oracle Database Performance Tuning Guide* to learn how to use index-organized tables to improve performance
- *Oracle Database SQL Language Reference* for CREATE TABLE ... ORGANIZATION INDEX syntax and semantics

Index-Organized Table Characteristics

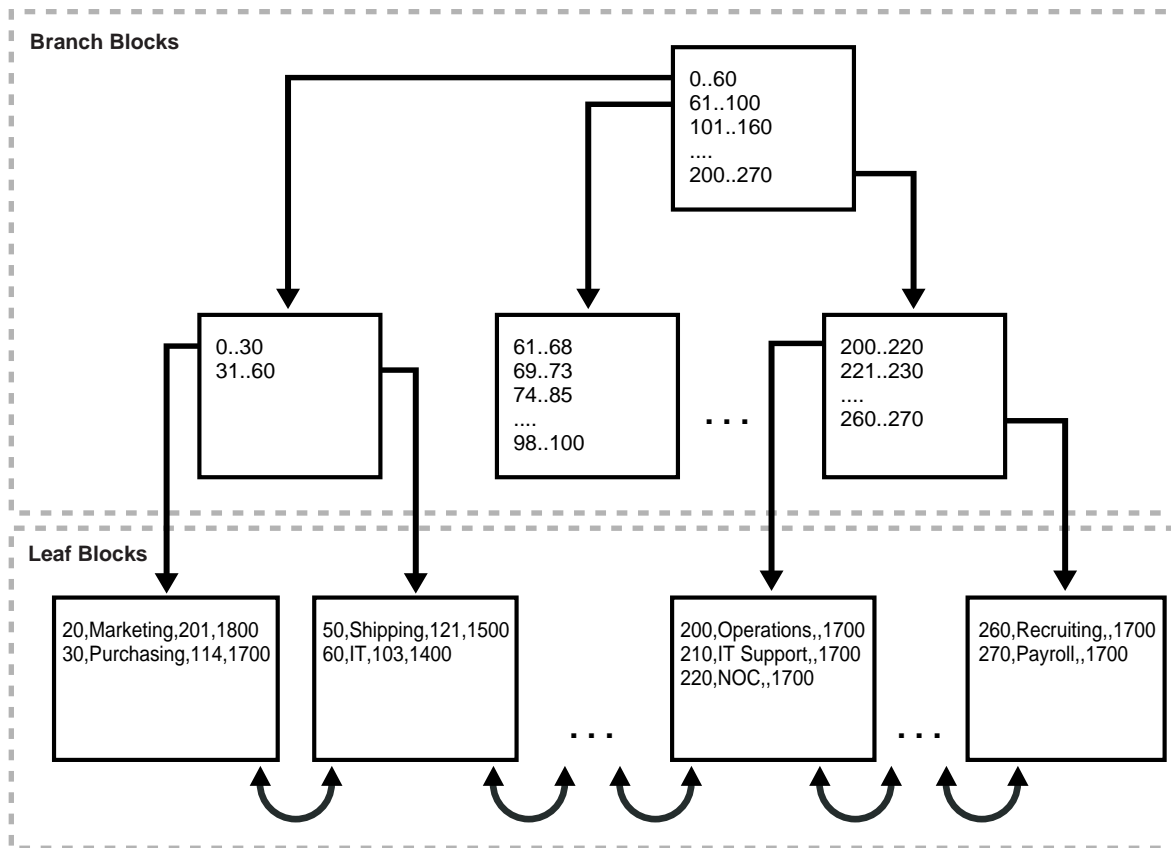
The database system performs all operations on index-organized tables by manipulating the B-tree index structure. [Table 3–4](#) summarizes the differences between index-organized tables and heap-organized tables.

Table 3–4 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 3–3](#) 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 3-3 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 3-3, 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

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,AAAPeXAAFAAAAyAAD
30,AAAPeXAAFAAAAyAAA
50,AAAPeXAAFAAAAyAAC
60,AAAPeXAAFAAAAyAAB
```

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"](#) on page 2-18
- ["Introduction to Logical Storage Structures"](#) on page 12-1

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, it is an index on an index. The secondary index is an independent schema object and is stored separately from the index-organized table.

As explained in ["Rowid Data Types"](#) on page 2-13, 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 (see ["Chained and Migrated Rows"](#) on page 12-16). 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.

See Also:

- *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 the 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

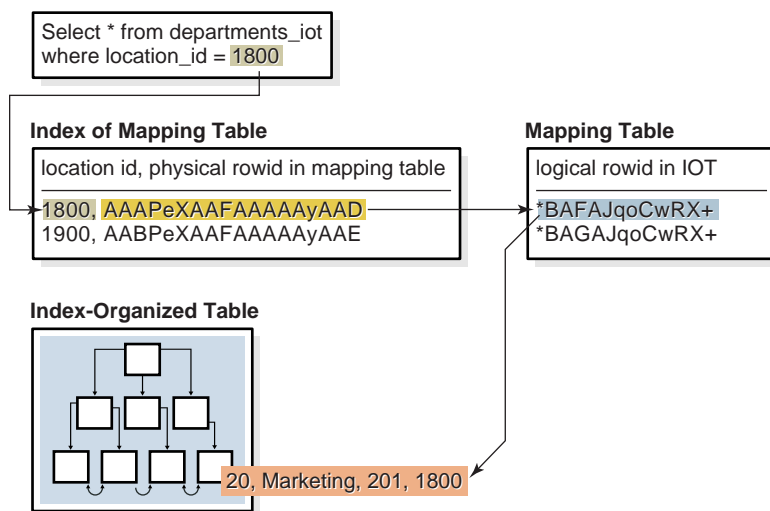
A secondary index on an index-organized table can be a **bitmap index**. As explained in "Bitmap Indexes" on page 3-13, a bitmap index stores a bitmap for each index key.

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.

Figure 3-4 illustrates index access for a query of the departments_iot table.

Figure 3-4 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" on page 2-19

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 Views](#)
- [Overview of Materialized Views](#)
- [Overview of Sequences](#)
- [Overview of Dimensions](#)
- [Overview of Synonyms](#)

Overview of Partitions

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. **DML** 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.

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, and all partitions in an index share the same indexed columns. 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.

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 strategy 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`.

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 want to populate a partitioned table with the sales rows shown in [Example 4-1](#).

Example 4–1 Sample Row Set for Partitioned Table

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

You create `time_range_sales` as a partitioned table using the statement in [Example 4–2](#). The `time_id` column is the partition key.

Example 4–2 Range-Partitioned Table

```
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 4–1](#). [Figure 4–1](#) 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.

Figure 4–1 Range Partitions

Table Partition SALES_1998						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
40	100530	30-NOV-98	9	33	1	44.99
125	9417	04-FEB-98	3	999	1	16.86
45	9491	28-AUG-98	4	350	1	47.45

Table Partition SALES_1999						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
116	11393	05-JUN-99	2	999	1	12.18
36	4523	27-JAN-99	3	999	1	53.89
24	11899	26-JUN-99	4	999	1	43.04

Table Partition SALES_2000						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
133	9450	01-DEC-00	2	999	1	31.28
35	2606	17-FEB-00	3	999	1	54.94

Table Partition SALES_2001						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
118	133	06-JUN-01	2	999	1	17.12
30	170	23-FEB-01	2	999	1	8.8

The range partition key value determines the high value of the range partitions, which is called the **transition point**. In [Figure 4–1](#), the SALES_1998 partition contains rows with partitioning key `time_id` values less than the transition point 01-JAN-1999.

The database creates **interval partitions** for data beyond that transition point. Interval partitions extend range partitioning by instructing the database to create partitions of the specified range or interval automatically when data inserted into the table exceeds all of the range partitions. In [Figure 4–1](#), the SALES_2001 partition contains rows with partitioning key `time_id` values greater than or equal to 01-JAN-2001.

List Partitioning In **list partitioning**, the database 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.

Assume that you create `list_sales` as a list-partitioned table using the statement in [Example 4–3](#). The `channel_id` column is the partition key.

Example 4-3 List-Partitioned Table

```

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 4-1](#). [Figure 4-2](#) 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.

Figure 4-2 List Partitions

Table Partition EVEN_CHANNELS						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
116	11393	05-JUN-99	2	999	1	12.18
118	133	06-JUN-01	2	999	1	17.12
133	9450	01-DEC-00	2	999	1	31.28
30	170	23-FEB-01	2	999	1	8.8
24	11899	26-JUN-99	4	999	1	43.04
45	9491	28-AUG-98	4	350	1	47.45

Table Partition ODD_CHANNELS						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
40	100530	30-NOV-98	9	33	1	44.99
36	4523	27-JAN-99	3	999	1	53.89
125	9417	04-FEB-98	3	999	1	16.86
35	2606	17-FEB-00	3	999	1	54.94

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. The hashing algorithm is designed to evenly distribute rows across devices so that each partition contains about the same number of rows.

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 statement in [Example 4-4](#). The `prod_id` column is the partition key.

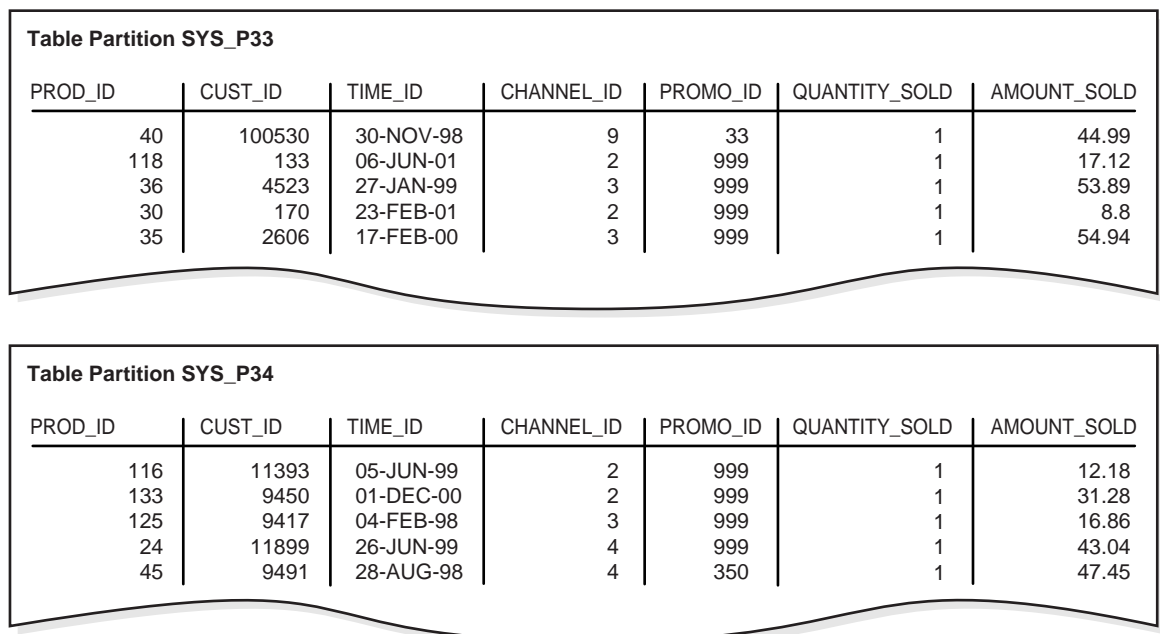
Example 4-4 Hash-Partitioned Table

```
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 4-1](#). [Figure 4-3](#) shows a possible row distribution in the two partitions. Note that 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. If you change the number of partitions, then the database redistributes the data over all of the partitions.

Figure 4-3 Hash Partitions



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

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. "[Partition Characteristics](#)" on page 4-2 gives examples of 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.

Some or all partitions of a heap-organized table can be stored in a compressed format. Compression saves space and can speed query execution. Thus, compression can be useful in environments such as data warehouses, where the amount of insert and update operations is small, and in **OLTP** environments.

The attributes for **table compression** can be declared 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.

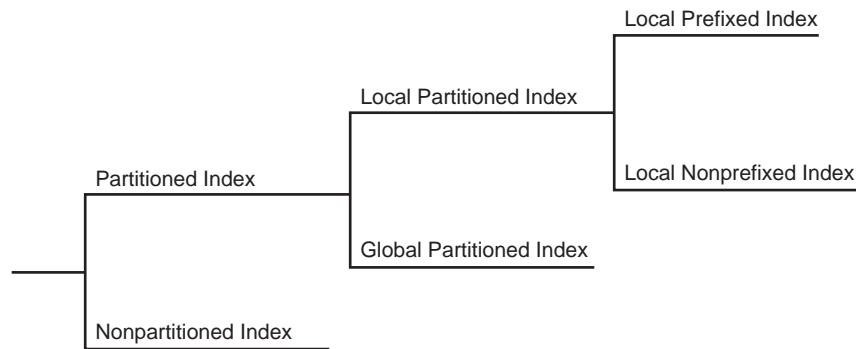
See Also:

- "[Table Compression](#)" on page 2-19 and "[Overview of Segments](#)" on page 12-21
- *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 decomposed 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.

**See Also:**

- ["Overview of Indexes"](#) on page 3-1
- *Oracle Database VLDB and Partitioning Guide* and *Oracle Database Performance Tuning 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 ["Data File Recovery"](#) on page 18-14). The entire index does not need to be rebuilt.

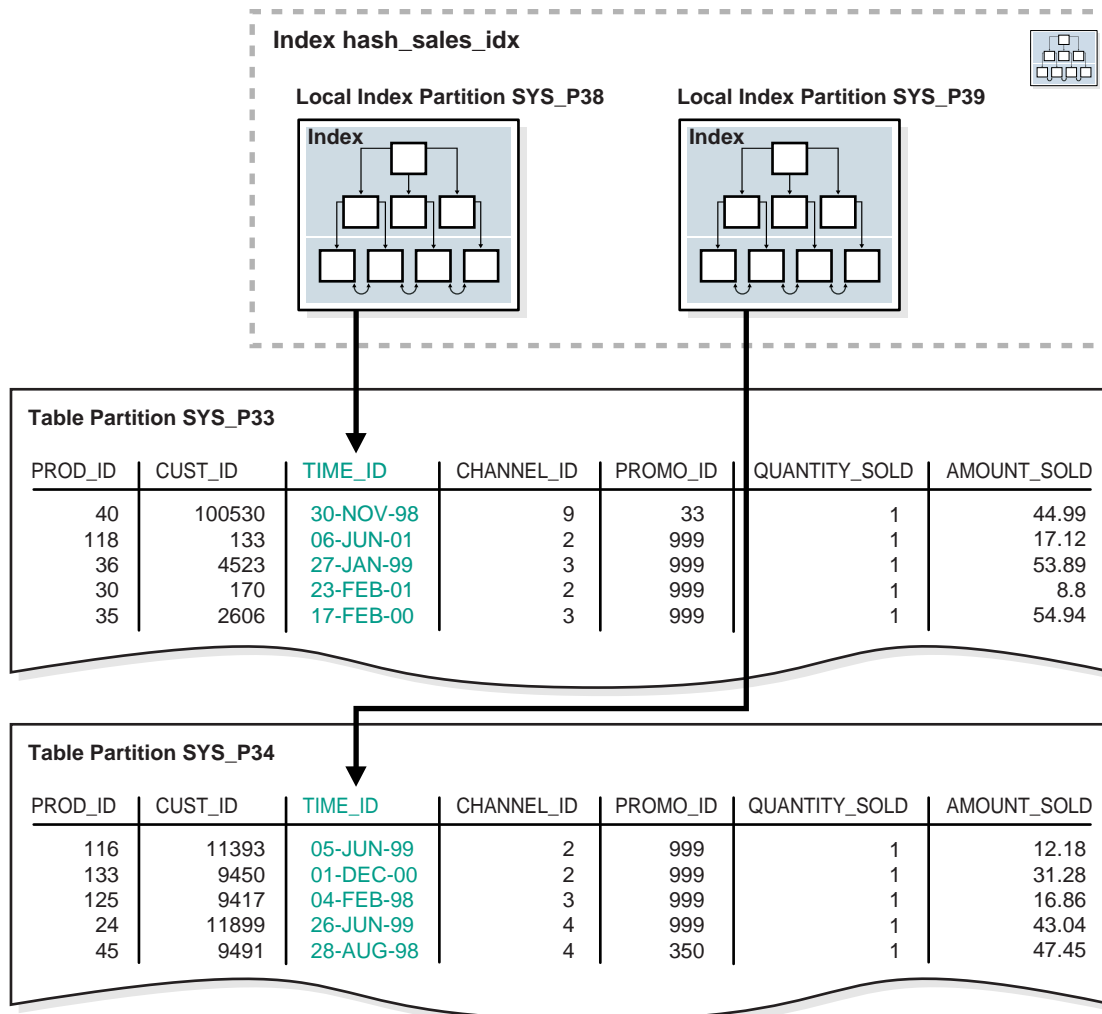
[Example 4-4](#) shows the creation statement for the partitioned `hash_sales` table, using the `prod_id` column as partition key. [Example 4-5](#) creates a local partitioned index on the `time_id` column of the `hash_sales` table.

Example 4-5 Local Partitioned Index

```
CREATE INDEX hash_sales_idx ON hash_sales(time_id) LOCAL;
```

In [Figure 4-4](#), 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 4–4 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 divided into the following subcategories:

- Local prefixed indexes

In this case, the partition keys are on the leading edge of the index definition. In [Example 4–2](#) on page 4-3, 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 [Example 4–5](#) on page 4-8, 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.

You can partition a global index by range or by hash. If partitioned by range, then the database partitions the global index on the ranges of values from the table columns you specify in the column list. If partitioned by hash, then the database assigns rows to the partitions using a hash function on values in the partitioning key columns.

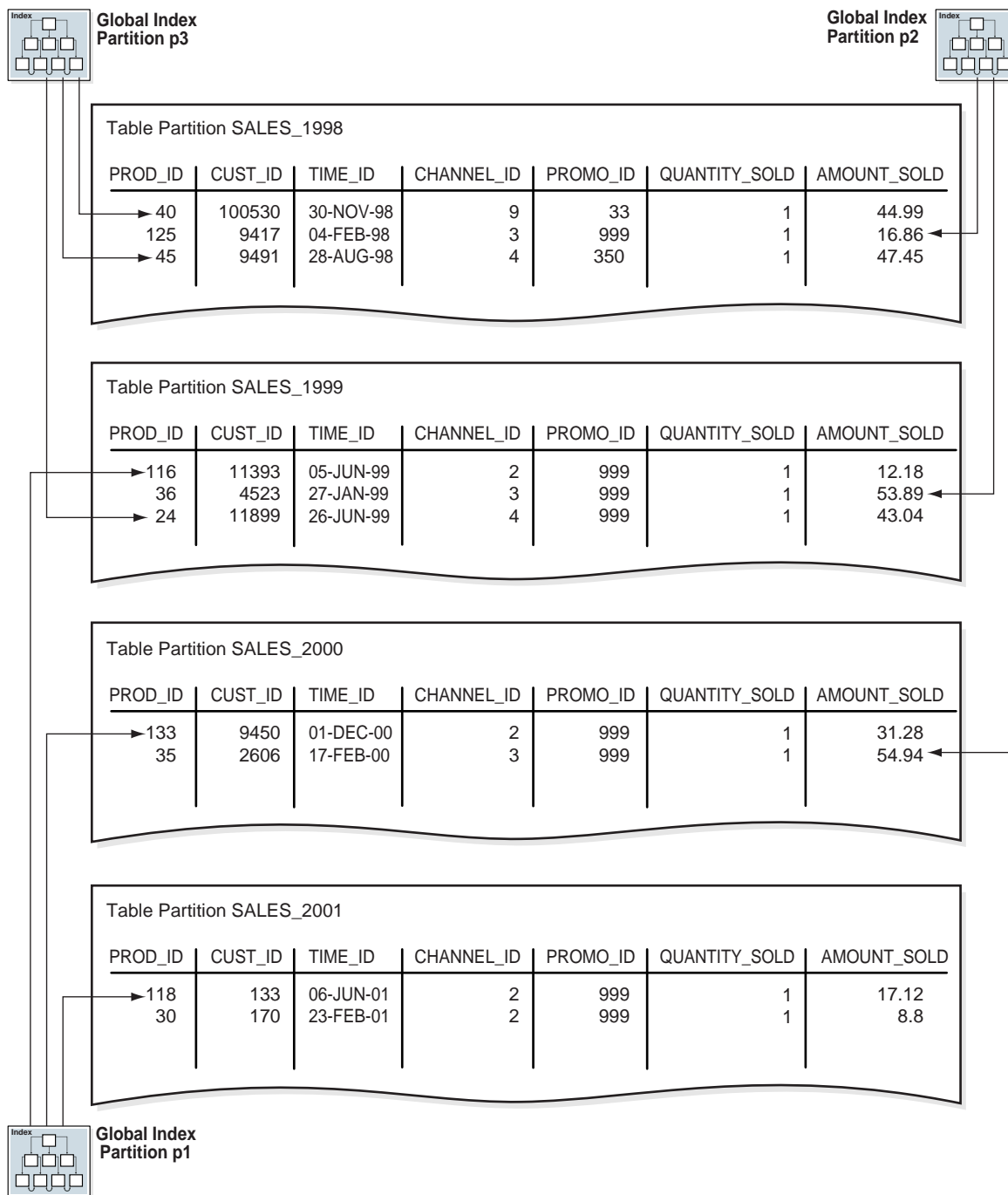
As an illustration, suppose that you create a global partitioned index on the `time_range_sales` table from [Example 4-2](#). In this table, rows for sales from 1998 are stored in one partition, rows for sales from 1999 are in another, and so on. [Example 4-6](#) creates a global index partitioned by range on the `channel_id` column.

Example 4-6 Global Partitioned Index

```
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 4-5](#), 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 4–5 Global Partitioned Index

**See Also:**

- *Oracle Database VLDB and Partitioning Guide* to learn how to use global partitioned indexes
- *Oracle Database SQL Language Reference* for CREATE INDEX ... GLOBAL examples

Partitioned Index-Organized Tables

You can partition an **index-organized table** (IOT) by range, list, or hash. Partitioning is useful for providing improved manageability, availability, and performance for IOTs. In addition, data cartridges that use IOTs can take advantage of the ability to partition their stored data.

Note the following characteristics of partitioned IOTs:

- Partition columns must be a subset of primary key columns.
- Secondary indexes can be partitioned locally and globally.
- OVERFLOW data segments are always equipartitioned with the table partitions.

Oracle Database supports bitmap indexes on partitioned and nonpartitioned index-organized tables. A mapping table is required for creating bitmap indexes on an index-organized table.

See Also: ["Overview of Index-Organized Tables"](#) on page 3-20

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. See ["Overview of Materialized Views"](#) on page 4-16.

Views enable you to tailor the presentation of data to different types of users. Views are often used to:

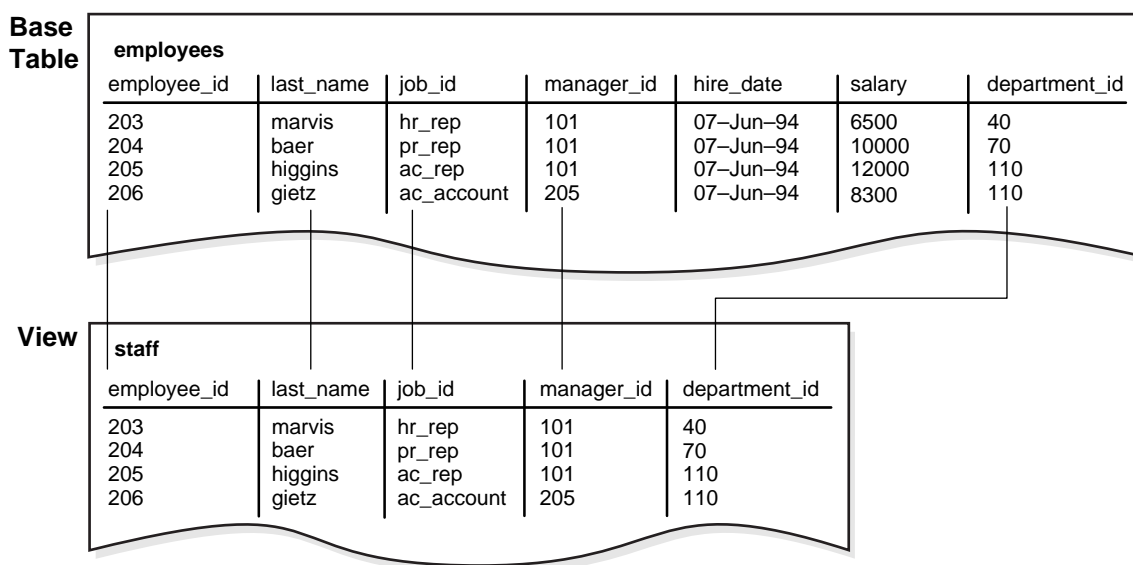
- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table
For example, [Figure 4-6](#) shows how the `staff` view does not show the `salary` or `commission_pct` columns of the base table `employees`.
- Hide data complexity
For example, 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. [Figure 4–6](#) graphically illustrates the view named `staff`. Notice that the view shows only five of the columns in the base table.

Figure 4–6 View



See Also:

- *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. For example, a view can contain up to 1000 columns, just like a table. 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  departments.department_id = 10;
```

A user executes the following query of `employees_view`:

```
SELECT last_name
FROM   employees_view
WHERE  employee_id = 9876;
```

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 = 9876;

```

See Also:

- ["Overview of the Optimizer"](#) on page 7-10 and *Oracle Database Performance Tuning Guide* to learn about query optimization
- ["Shared SQL Areas"](#) on page 14-16

Updatable Join Views

A **join view** is defined as a view that has multiple tables or views in its `FROM` clause. In [Example 4-7](#), the `staff_dept_10_30` view joins the `employees` and `departments` tables, including only employees in departments 10 or 30.

Example 4-7 Join View

```

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 view created in [Example 4-7](#) 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

All updatable columns of a join view must map to columns of a key-preserved table. A **key-preserved table** in a join query is a table in which each row of the underlying table appears at most one time in the output of the query. In [Example 4-7](#), `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.

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 was stored as an object type. You can also define views with columns that are object data types, such as objects, *REFs*, and collections (nested tables and *VARRAYS*).

Like relational views, object views can present only the data that you 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` command

Overview of Materialized Views

Materialized views are query results that have been stored or "materialized" in advance as schema objects. The `FROM` clause of the query can name tables, views, and materialized views. Collectively these objects are called **master tables** (a replication term) or **detail tables** (a data warehousing term).

Materialized views are used to summarize, compute, replicate, and distribute data. They are suitable in various computing environments, such as the following:

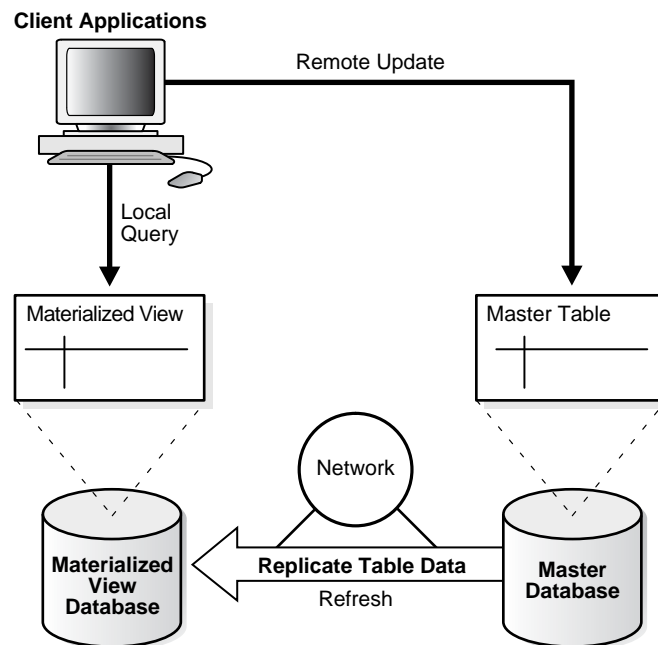
- In data warehouses, you can use materialized views to compute and store data generated from aggregate functions such as sums and averages.

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 (see "[Data Warehouse Architecture \(Basic\)](#)" on page 17-16). You can also use materialized views to compute joins with or without aggregations. If compatibility is set to Oracle9i or higher, then materialized views are usable for queries that include filter selections.

- In materialized view **replication**, 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, you can use materialized views to 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 4-7](#) 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 4-7 Materialized View



See Also:

- ["Information Sharing"](#) on page 17-21 to learn about replication with Oracle Streams
- *Oracle Database 2 Day + Data Replication and Integration Guide* and *Oracle Database Advanced Replication* to learn how to use materialized views
- *Oracle Database SQL Language Reference* to learn about the `CREATE MATERIALIZED VIEW` statement

Characteristics of Materialized Views

Materialized views share some characteristics of nonmaterialized views and indexes. 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 their master tables. The refresh method can be incremental, known as **fast refresh**, or a **complete refresh**.

A complete refresh occurs when the materialized view is initially defined as `BUILD IMMEDIATE`, unless the materialized view references a prebuilt table. The refresh involves executing the query that defines the materialized view. This process can be slow, especially if the database must read and process huge amounts of data.

A fast refresh eliminates the need to rebuild materialized views from scratch. Thus, processing only the changes can result in a very fast refresh time. Materialized views can be refreshed either on demand or at regular time intervals. Alternatively, materialized views in the same database as their master tables can be refreshed whenever a transaction commits its changes to the master tables.

For materialized views that use the fast refresh method, a **materialized view log** or **direct loader log** keeps a record of changes to the master tables. A materialized view

log is a schema object that records changes to master table data so that a materialized view defined on the master table can be refreshed incrementally. Each materialized view log is associated with a single master table. The materialized view log resides in the same database and schema as its master table.

See Also:

- *Oracle Database Data Warehousing Guide* to learn how to refresh materialized views
- *Oracle Database Advanced Replication* to learn about materialized view logs

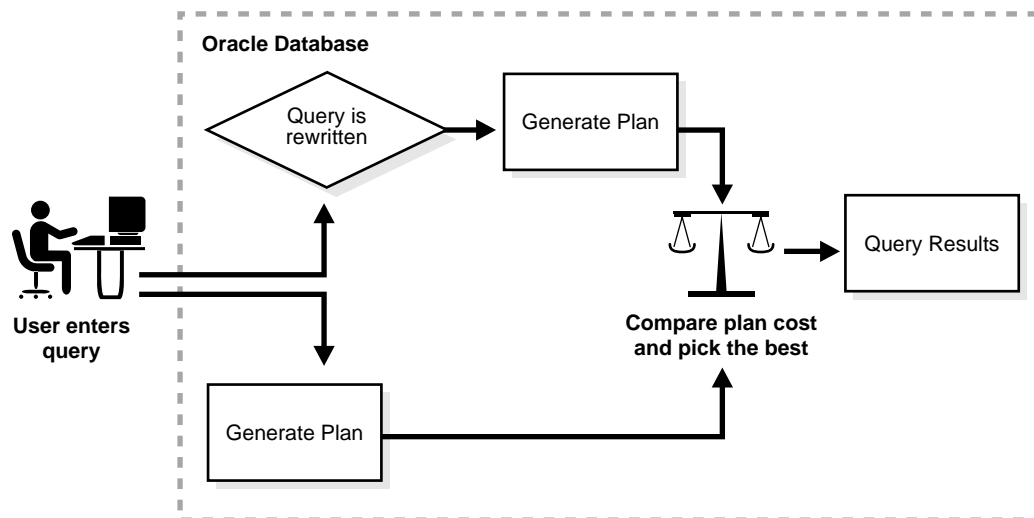
Query Rewrite

Query rewrite is an optimization technique that 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 **optimizer 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. [Figure 4–8](#) shows the database generating an **execution plan** for the original and rewritten query and choosing the lowest-cost plan.

Figure 4–8 Query Rewrite



See Also:

- ["Overview of the Optimizer"](#) on page 7-10
- *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, such as the following:

- The name of the sequence
- Whether the sequence ascends or descends
- 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* and *Oracle Database Administrator's Guide* to learn how to manage sequences
- *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. In this way, the database can generate 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.

Caution: 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: [Chapter 9, "Data Concurrency and Consistency"](#)

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 2007 and 2008.
- Create a cross-tabular analysis of our operations showing expenses by territory in South America for 2007 and 2008. Include all possible subtotals.
- List the top 10 sales representatives in Asia according to 2008 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.

See Also: ["Overview of Data Warehousing and Business Intelligence"](#) on page 17-14

Hierarchical Structure of a Dimension

A **dimension table** is a logical structure that defines hierarchical relationships between pairs of columns or column sets. 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**.

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.

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

Dimensions are created with SQL statements. The `CREATE DIMENSION` 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 Warehouse Builder Data Modeling, ETL, and Data Quality Guide* for information about how dimensions are used in a warehousing environment
- *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.

In [Example 4-9](#), 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.

Example 4-8 Public 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 [Example 4-9](#), 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.

Example 4-9 Public Synonym

```
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

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](#)

See Also: ["Overview of Tables"](#) on page 2-6

Introduction to Data Integrity

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. It is important that data maintain **data integrity**, which is adherence to these rules, as determined by the database administrator or application developer.

Techniques for Guaranteeing Data Integrity

When designing a database application, developers have various options for guaranteeing the integrity of data stored in the database. These options include:

- Enforcing business rules with triggered stored database procedures, as described in ["Overview of Triggers"](#) on page 8-16
- Using stored procedures to completely control access to data, as described in ["Introduction to Server-Side Programming"](#) on page 8-1
- 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

This chapter explains the basic concepts of integrity constraints.

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 unless it is not 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** (see "[Overview of the Data Dictionary](#)" on page 6-1). 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:

- *Oracle Database 2 Day Developer's Guide* and *Oracle Database 2 Day Developer's Guide* to learn how to maintain data integrity
- *Oracle Database 2 Day DBA* and *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 called an **inline** specification. A constraint specified as part of the table definition is called an **out-of-line** specification.

The term **key** is used in the definitions of several types of integrity constraints. 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**.

[Table 5-1](#) describes the types of constraints. Each can be specified either inline or out-of-line, except for NOT NULL, which must be inline.

Table 5-1 Types of 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" on page 5-3
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" on page 5-3
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" on page 5-5

Table 5–1 (Cont.) Types of Constraints

Constraint Type	Description	See Also
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 " on page 5-6
Check	Requires a database value to obey a specified condition.	" Check Constraints " on page 5-9
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.	<i>Oracle Database Object-Relational Developer's Guide</i> to learn about REF constraints

See Also:

- "[Overview of Tables](#)" on page 2-6
- *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 `last_name` column. An attempt to insert an employee row without a last name 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"."LAST_NAME")
```

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 Advanced Application Developer's 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 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 2–1](#) on page 2-8, 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 [Example 5–1](#).

Example 5–1 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
```

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.

See Also:

- ["Unique and Nonunique Indexes"](#) on page 3-4
- *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, surrogate keys are generated by a **sequence**.

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.

Example 5–1 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', 'ICHAN', 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 a new 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 Advanced Application Developer's 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**. 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. An example of a referential integrity rule is an employee can work for only an existing department.

Table 5–2 lists terms associated with referential integrity constraints.

Table 5–2 Referential Integrity Constraint Terms

Term	Definition
Foreign key	<p>The column or set of columns included in the definition of the constraint that reference a referenced key. For example, the <code>department_id</code> column in <code>employees</code> is a foreign key that references the <code>department_id</code> column in <code>departments</code>.</p> <p>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.</p> <p>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.</p>
Referenced key	The unique key or primary key of the table referenced by a foreign key. For example, the <code>department_id</code> column in <code>departments</code> is the referenced key for the <code>department_id</code> column in <code>employees</code> .
Dependent or child table	The table that includes the foreign key. This table is dependent on the values present in the referenced unique or primary key. For example, the <code>employees</code> table is a child of <code>departments</code> .
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 <code>departments</code> table is a parent of <code>employees</code> .

Figure 5–1 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 5–1 Referential Integrity Constraints

Parent Key
Primary key of
referenced table

Referenced or Parent Table

Table DEPARTMENTS			
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
60	IT	103	1400
90	Executive	100	1700

Foreign Key
(values in dependent
table must match a
value in unique key
or primary key of
referenced table)

Dependent or Child Table

Table EMPLOYEES						
EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	MANAGER_ID	DEPARTMENT_ID
100	King	SKING	17-JUN-87	AD_PRES		90
101	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	100	90
102	De Hann	LDEHANN	13-JAN-93	AD_VP	100	90
103	Hunold	AHUNOLD	03-JAN-90	IT_PROG	102	60

This row violates the referential
constraint because "99" is not
present in the referenced table's
primary key; therefore, the row
is not allowed in the table.

INSERT
INTO

207	Ashdown	AASHDOWN	17-DEC-07	MK_MAN	100	99
208	Green	BGREEN	17-DEC-07	AC_MGR	101	

This row is allowed in the table
because a null value is entered
in the DEPARTMENT_ID column;
however, if a not null constraint
is also defined for this column,
this row is not allowed.

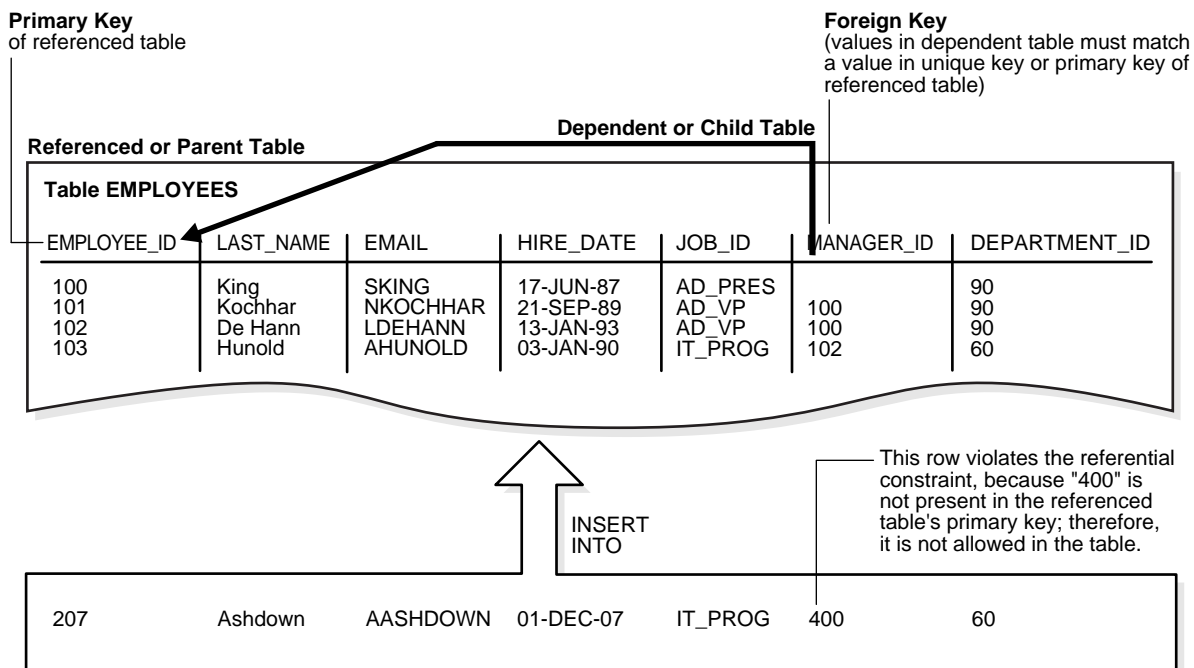
See Also: *Oracle Database 2 Day Developer's Guide* and *Oracle Database Advanced Application Developer's Guide* to learn how to add foreign key constraints to a table

Self-Referential Integrity Constraints

Figure 5–2 shows a **self-referential integrity constraint**. In this case, a foreign key references a parent key in the same table.

In Figure 5–2, the referential integrity 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 5–2 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 user could insert a row into `hr.employees` without specifying 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.

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 5–3](#) 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 5–3 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"](#) on page 8-16.

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"](#) on page 9-21 and ["Overview of Indexes"](#) on page 3-1

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. 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 **validate** (`VALIDATE`) or not validate (`NOVALIDATE`) existing data. If `VALIDATE` is specified, then 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. If `NOVALIDATE` is specified, then existing data need not conform to the constraint.

The behavior of `VALIDATE` and `NOVALIDATE` always depends on whether the constraint is enabled or disabled. [Table 5–4](#) summarizes the relationships.

Table 5–4 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. Thus, existing rows can violate the constraint, but new or modified rows must conform to the rules.

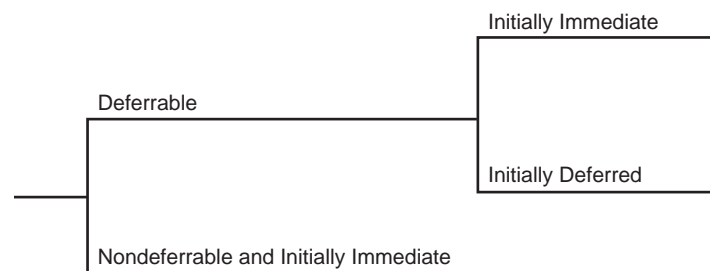
Table 5–4 (Cont.) Checks on Modified and Existing Data

Modified Data	Existing Data	Summary
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

Deferrable Constraints

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 depicts the options for deferrable constraints.



Nondeferrable Constraints

If a constraint is not deferrable, then 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, assume that you create a nondeferrable `NOT NULL` constraint for the `employees.last_name` column. If a user 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 lets you disable the constraint until all the 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

Some examples may help illustrate when Oracle Database performs the checking of constraints. Assume the following:

- The `employees` table has the structure shown in [Figure 5–2](#) on page 5-8.
- The self-referential constraint makes entries in the `manager_id` column dependent on the values of the `employee_id` column.

Insertion of a Value in a Foreign Key Column When No Parent Key Value Exists

Consider 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 any existing value in the `employee_id` column? Some possibilities are:

- A null can be entered for the `manager_id` column of the first row, if the `manager_id` column does not have a `NOT NULL` constraint defined on it.

Because nulls are allowed in foreign keys, this row is inserted into the table.

- The same value can be entered 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 has been completely run. To allow a row to be entered with the same values in the parent key and the foreign key, the database must first run the statement (that is, 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` statement, 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 statement. All rows are inserted first, and then all rows are checked 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.

An Update of All Foreign Key and Parent Key Values

Consider the same self-referential integrity constraint in a different scenario. 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. Because manager numbers are really employee numbers (see [Figure 5–3](#)), the manager numbers must also increase by 5000.

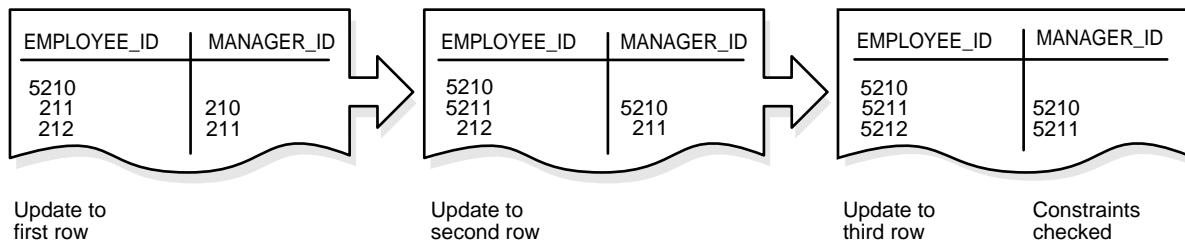
Figure 5–3 The employees Table Before Updates

EMPLOYEE_ID	MANAGER_ID
210	
211	210
212	211

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 legal because the database effectively checks constraints after the statement completes. [Figure 5–4](#) shows that the database performs the actions of the entire SQL statement before checking constraints.

Figure 5–4 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 same mechanism is used 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.

Data Dictionary and Dynamic Performance Views

This chapter describes the central set of read-only reference tables and views of each Oracle database, known collectively as the **data dictionary**. The chapter also describes the **dynamic performance views**, which 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. A 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 (see "[User Accounts](#)" on page 17-1)

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 (see "[Data Definition Language \(DDL\) Statements](#)" on page 7-3)

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](#)" on page 2-1

Contents of the Data Dictionary

The data dictionary consists of the following types of objects:

- Base tables

These underlying tables 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 views decode the base table data into useful information, such as user or table names, using **joins** and **WHERE** clauses to simplify the information. These 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 [Table 6-1](#). By querying the appropriate views, you can access only the information relevant for you.

Table 6-1 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

.
.
.
```

See Also:

- *Oracle Database Reference* for a complete list of data dictionary views and their columns
- ["Overview of Views"](#) on page 4-12

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.

For example, the following 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.

See Also: ["PL/SQL Subprograms"](#) on page 8-3

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

Storage of the Data Dictionary

The data dictionary base tables are the first objects created in any Oracle database. All data dictionary tables and views for a database are stored in the SYSTEM tablespace. Because the SYSTEM tablespace is always online when the database is open, the data dictionary is always available when the database is open.

See Also: "[The SYSTEM Tablespace](#)" on page 12-32 for more information about the SYSTEM tablespace

How Oracle Database Uses the Data Dictionary

The Oracle Database user SYS owns all base tables and user-accessible views of the data dictionary. Data in the base tables of the data dictionary *is necessary for Oracle Database to function*. Therefore, 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.

Caution: Altering or manipulating the data in data dictionary tables can permanently and detrimentally affect database operation.

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 also 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 new rows are added 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.

See Also: ["SYS and SYSTEM Schemas"](#) on page 2-5

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. Users should avoid naming their own schema objects with the same names as those used for public synonyms.

See Also: ["Overview of Synonyms"](#) on page 4-22

Cache the Data Dictionary for Fast Access

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. Parsing information is typically kept in the caches. 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"](#) on page 14-19

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. Application developers who write programs that refer to the data dictionary should refer to the public synonyms rather than the underlying tables. Synonyms are less likely to change between releases.

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 called **dynamic performance views** because they are continuously updated while a database is open and in use. The views, also sometimes called **V\$ 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 (see ["Oracle Enterprise Manager"](#) on page 18-2).

- Administrators can use the views for performance monitoring and debugging.

See Also: *Oracle Database Reference* for a complete list of the dynamic performance views

Contents of the Dynamic Performance Views

Dynamic performance views are sometimes 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 (see "[Database Server Grid](#)" on page 17-12).

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:

- "[Tools for Database Installation and Configuration](#)" on page 18-4 to learn about DBCA
- *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. Thus, they 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 is dependent on the state of the database and 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: [Chapter 9, "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**. Two styles of interface are provided: 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 II

Oracle Data Access

This part contains the following chapters:

- [Chapter 7, "SQL"](#)
- [Chapter 8, "Server-Side Programming: PL/SQL and Java"](#)

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. Although some Oracle tools and applications mask SQL use, all database operations 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.

See Also:

- [Oracle Database SQL Language Reference](#) for an introduction to SQL
- ["Introduction to Server-Side Programming"](#) on page 8-1 and ["Client-Side Database Programming"](#) on page 19-5

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. The SQL language compiler performs the work of generating a procedure to navigate the database and perform the desired task.

SQL enables you to work with data at the logical level. You need be concerned with implementation details only when you want to manipulate the data. For example, the following statement queries records for employees whose last name begins with K:

```
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. These rows can be passed as a unit to the user, to another SQL statement, or to an application. You do not need to process the rows one by one, nor are you required to know how the rows are physically stored or retrieved.

All SQL statements use the **optimizer**, a part of Oracle Database that determines the most efficient means of accessing the specified 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 latest SQL standard was adopted in July 2003 and is often called SQL:2003. One part of the SQL standard, Part 14, SQL/XML (ISO/IEC 9075-14) was revised in 2006 and is often referred to as SQL/XML:2006.

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 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
- "[Tools for Database Administrators](#)" on page 18-2 and "[Tools for Database Developers](#)" on page 19-1

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 (**DDL**) statements define, structurally change, and drop **schema objects**. For example, 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`).

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.

[Example 7-1](#) 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 privileges on this table to a user, and then drop the table.

Example 7-1 DDL Statements

```
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 SELECT ON plants TO scott;

REVOKE SELECT 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 [Example 7-1](#), 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:

- ["Overview of Database Security"](#) on page 17-1 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 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 enable you to change the structure of the database, DML statements enable you to 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). See ["How Oracle Database Processes DML"](#) on page 7-22.
- 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.

See Also:

- ["Introduction to Transactions"](#) on page 10-1
- *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**.

[Table 7-1](#) 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 7-1 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. 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

Table 7–1 (Cont.) Keywords in a SQL Statement

Keyword	Required?	Description	Capability
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. [Example 7–2](#) 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.

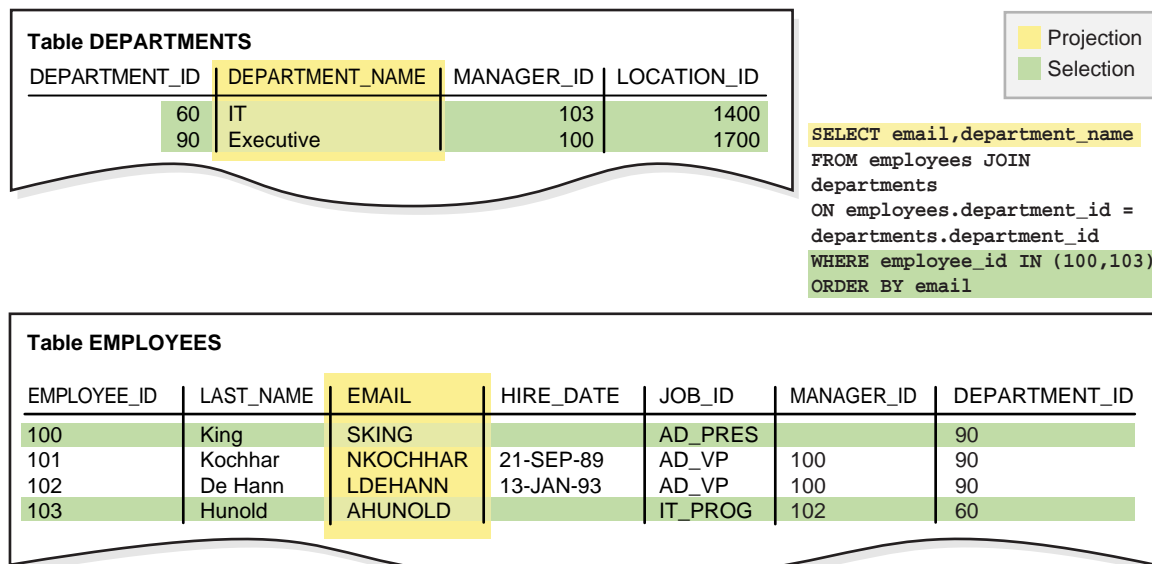
Example 7–2 Sample Join

```
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

[Figure 7–1](#) graphically represents the operations of projection and selection in the join shown in [Example 7–2](#).

Figure 7–1 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. For example, a **left outer join** of `employees` and `departments` retrieves all rows in the `employees` table even if there is no match in `departments`. A **right outer join** retrieves all rows in `departments` even if there is no match in `employees`.

- Cartesian products

If two tables in a join query have no join condition, then the database returns their **Cartesian product**. 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 Language Reference* for detailed descriptions and examples of joins

Subqueries and Implicit Queries

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 [Example 7-3](#), the subquery in parentheses is the **inner query block**. 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.

Example 7-3 Subquery

```
SELECT first_name, last_name
FROM   employees
WHERE  department_id
IN     (SELECT department_id FROM departments WHERE location_id = 1800);
```

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 decides to query the `employees` first and then obtain the department IDs. The optimizer determines the best sequence of steps to retrieve the requested rows.

An **implicit query** is a component of a DML statement that retrieves data without using a subquery. An `UPDATE`, `DELETE`, or `MERGE` statement that does not explicitly include a `SELECT` statement uses an implicit query to retrieve rows to be modified. For example, the following statement includes an implicit query for the Baer record:

```
UPDATE employees
```

```
SET salary = salary*1.1
WHERE last_name = 'Baer';
```

The only DML statement that does *not* necessarily include a query component is an `INSERT` statement with a `VALUES` clause. For example, an `INSERT INTO TABLE mytable VALUES (1)` statement does not retrieve rows before inserting a row.

See Also: ["Virtual Private Database \(VPD\)"](#) on page 17-4

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` command 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"](#) on page 10-1
- ["Deferrable Constraints"](#) on page 5-11
- *Oracle Database SQL Language Reference*

Session Control Statements

Session control statements dynamically manage the properties of a user **session**. As explained in ["Connections and Sessions"](#) on page 15-4, 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 enabling and disabling SQL tracing (`ALTER SESSION`).
- Enable and disable roles, which are groups of privileges, for the current session (`SET ROLE`).

The following example turns on SQL tracing for the session and then enables all roles granted in the current session except `dw_manager`:

```
ALTER SESSION SET SQL_TRACE = TRUE;
```

```
SET ROLE ALL EXCEPT dw_manager;
```

Session control statements do not implicitly commit the current transaction.

See Also: *Oracle Database SQL Language Reference* for `ALTER SESSION` syntax and semantics

System Control Statement

System control statements change 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.

Following are examples of system control statements:

```
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. They 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 **cursors** (`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"](#) on page 8-1 and ["Client-Side APIs"](#) on page 19-7

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

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.

Many different ways of processing a DML statement are often possible. For example, the order in which tables or indexes are accessed can vary. The steps that the database uses to execute a statement greatly affect how quickly the statement runs. 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, including query conditions, available **access paths**, statistics gathered for the system, and hints. For any SQL statement processed by Oracle, 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.

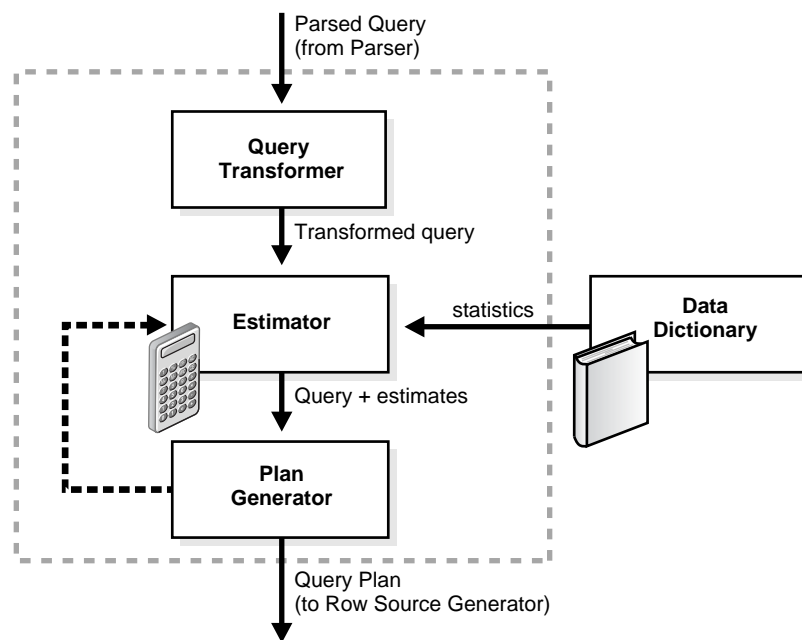
See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about using `DBMS_STATS`
- *Oracle Database Performance Tuning Guide* for more information about the optimizer and using hints

Optimizer Components

The optimizer contains three main components, which are shown in [Figure 7-2](#).

Figure 7-2 Optimizer Components



The input to the optimizer is a parsed query (see ["SQL Parsing"](#) on page 7-16). 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 (see ["SQL Row Source Generation"](#) on page 7-19).

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 is represented by a set of query blocks.

See Also: ["Query Rewrite"](#) on page 4-19

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 and picks the plan with the lowest cost. The optimizer generates subplans for each of the nested subqueries and unmerged views, which is represented by 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 optimizer automatically manages plans and ensures that only verified plans are used. SQL Plan Management (SPM) allows controlled plan evolution by only using a new plan after it has been verified to be perform better than the current plan.

Diagnostic tools such as the `EXPLAIN PLAN` statement enable you to view execution plans chosen by the optimizer. `EXPLAIN PLAN` shows the query plan for the specified SQL query if it were executed now in the current session. Other diagnostic tools are Oracle Enterprise Manager and the SQL*Plus `AUTOTRACE` command. [Example 7-6](#) on page 7-20 shows the execution plan of a query when `AUTOTRACE` is enabled.

See Also:

- ["Tools for Database Administrators"](#) on page 18-2
- *Oracle Database SQL Language Reference* to learn about `EXPLAIN PLAN`
- *Oracle Database Performance Tuning Guide* to learn about the optimizer components

Access Paths

An **access path** is the way in which data is retrieved from the database. 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** that separates used from unused space (see "[Segment Space and the High Water Mark](#)" on page 12-27).

- 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](#)" on page 3-6). 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 is used to retrieve 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](#)" on page 2-23). 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 is used to locate 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](#)" on page 2-25). 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 Performance Tuning Guide* to learn about access paths

Optimizer Statistics

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.

Optimizer statistics are created for the purposes of query optimization and are stored in the data dictionary. These statistics should not be confused with performance statistics visible through dynamic performance views.

See Also:

- *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database Performance 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.

For example, 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 [Example 7-4](#).

Example 7-4 Execution Plan for `SELECT` with `FIRST_ROWS` Hint

```
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		

The execution plan in [Example 7-4](#) 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 7-5](#) shows the same statement, but without the optimizer hint.

Example 7-5 Execution Plan for `SELECT` with No 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	

The execution plan in [Example 7-5](#) joins two indexes to return the requested records as fast as possible. Rather than repeatedly going from index to table as in [Example 7-4](#), 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 7-4](#), which accesses the table by index rowid, the plan in [Example 7-5](#) 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 Performance Tuning Guide* to learn how to use optimizer hints

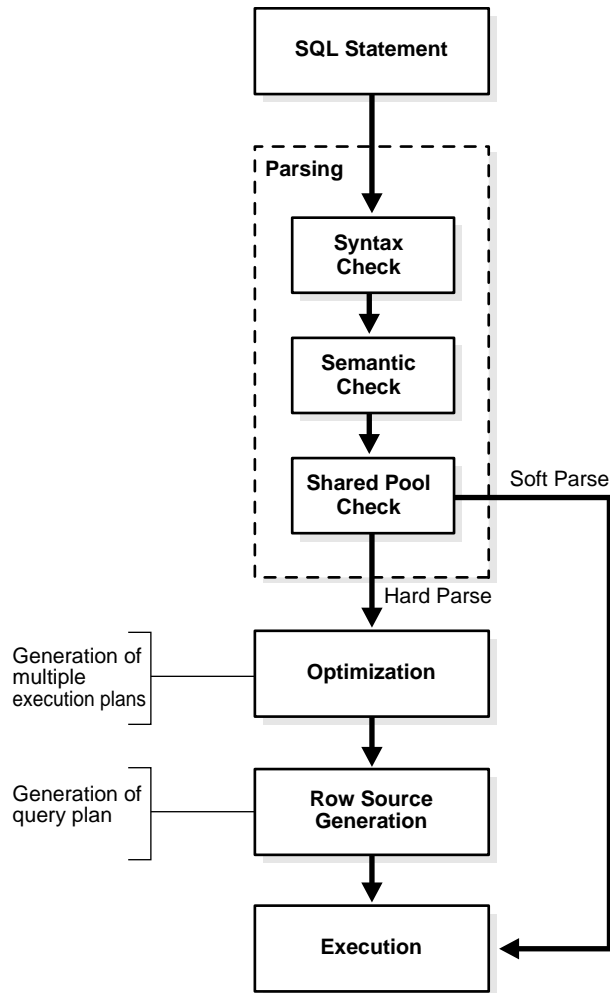
Overview of SQL Processing

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

[Figure 7-3](#) depicts the general stages of SQL processing: parsing, optimization, row source generation, and execution. Depending on the statement, the database may omit some of these steps.

Figure 7-3 Stages of SQL Processing



SQL Parsing

As shown in Figure 7-3, the first stage of SQL processing is **parsing**. This stage involves separating the pieces of a SQL statement into a data structure that can be processed by other routines. The database parses a statement when instructed by the application, which means that only the application, and not the database itself, can reduce the number of parses.

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

deadlocks or errors in data conversion only during statement execution (see "Locks and Deadlocks" on page 9-16).

Syntax Check Oracle Database must check each SQL statement for syntactic validity. A statement that breaks a rule for well-formed SQL syntax fails the check. For example, the following statement fails because the keyword FROM is misspelled as FORM:

```
SQL> SELECT * FORM employees;
SELECT * FORM employees
      *
ERROR at line 1:
ORA-00923: FROM keyword not found where expected
```

Semantic Check The semantics of a statement are its meaning. Thus, a semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist. A syntactically correct statement can fail a semantic check, as shown in the following example of a query of a nonexistent table:

```
SQL> SELECT * FROM nonexistent_table;
SELECT * FROM nonexistent_table
      *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Shared Pool Check During the parse, the database performs a **shared pool check** to determine whether it can skip resource-intensive steps of statement processing. To this end, the database uses a **hashing** algorithm to generate a hash value for every SQL statement. The statement hash value is the **SQL ID** shown in V\$SQL.SQL_ID.

When a user submits a SQL statement, the database searches the **shared SQL area** to see if an existing parsed statement has the same hash value. The hash value of a SQL statement is distinct from the following values:

- Memory address for the statement

Oracle Database uses the SQL ID to perform a keyed read in a lookup table. In this way, the database obtains possible memory addresses of the statement.

- Hash value of an execution plan for the statement

A SQL statement can have multiple plans in the shared pool. Each plan has a different hash value. If the same SQL ID has multiple plan hash values, then the database knows that multiple plans exist for this SQL ID.

Parse operations fall into the following categories, depending on the type of statement submitted and the result of the hash check:

- Hard parse

If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code. This operation is known as a **hard parse**, or a **library cache miss**. The database always perform a hard parse of DDL.

During the hard parse, the database accesses the **library cache** and **data dictionary cache** numerous times to check the data dictionary. When the database accesses these areas, it uses a serialization device called a **latch** on required objects so that their definition does not change (see "Latches" on page 9-25). Latch contention increases statement execution time and decreases concurrency.

- Soft parse

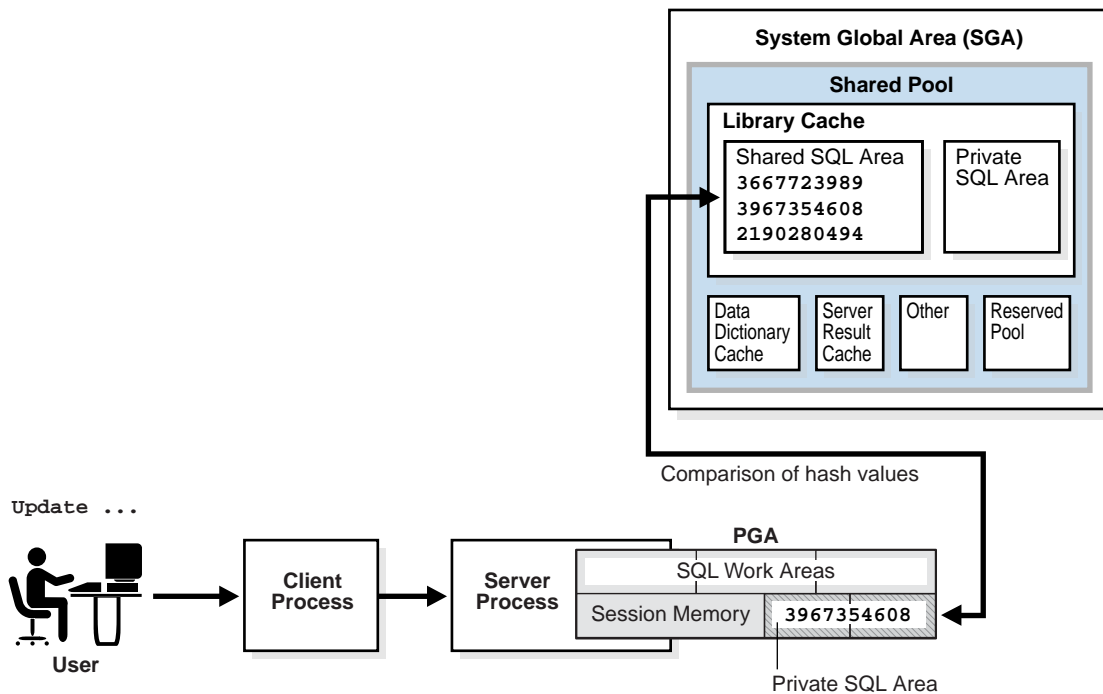
A **soft parse** is any parse that is not a hard parse. If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code. This reuse of code is also called a **library cache hit**.

Soft parses can vary in the amount of work they perform. For example, configuring the session shared SQL area can sometimes reduce the amount of latching in the soft parses, making them "softer."

In general, a soft parse is preferable to a hard parse because the database skips the optimization and row source generation steps, proceeding straight to execution.

Figure 7–4 is a simplified representation of a shared pool check of an UPDATE statement in a **dedicated server** architecture.

Figure 7–4 Shared Pool Check



If a check determines that a statement in the shared pool has the same hash value, then the database performs semantic and environment checks to determine whether the statements mean the same. Identical syntax is not sufficient. For example, suppose two different users log in to the database and issue the following SQL statements:

```
CREATE TABLE my_table ( some_col INTEGER );
SELECT * FROM my_table;
```

The SELECT statements for the two users are syntactically identical, but two separate schema objects are named my_table. This semantic difference means that the second statement cannot reuse the code for the first statement.

Even if two statements are semantically identical, an environmental difference can force a hard parse. In this case, the environment is the totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings. Consider the following series of SQL statements executed by a single user:

```
ALTER SYSTEM FLUSH SHARED_POOL;
SELECT * FROM my_table;
```

```
ALTER SESSION SET OPTIMIZER_MODE=FIRST_ROWS;
SELECT * FROM my_table;
```

```
ALTER SESSION SET SQL_TRACE=TRUE;
SELECT * FROM my_table;
```

In the preceding example, the same `SELECT` statement is executed in three different optimizer environments. Consequently, the database creates three separate shared SQL areas for these statements and forces a hard parse of each statement.

See Also:

- ["Private SQL Area"](#) on page 14-5 and ["Shared SQL Areas"](#) on page 14-16
- *Oracle Database Performance Tuning Guide* to learn how to configure the shared pool

SQL Optimization

As explained in ["Overview of the Optimizer"](#) on page 7-10, 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: *Oracle Database Performance 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 iterative plan is a binary program that, when executed by the SQL virtual machine, produces the result set.

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.

The row source generator produces a **row source tree**, which is a collection of row sources. The row source tree shows the following information:

- An ordering of the tables referenced by the statement
- An access method for each table mentioned in the statement
- A join method for tables affected by join operations in the statement
- Data operations such as filter, sort, or aggregation

[Example 7-6](#) shows the execution plan of a `SELECT` statement when `AUTOTRACE` is enabled. The statement selects the last name, job title, and department name for all

employees whose last names begin with the letter A. The execution plan for this statement is the output of the row source generator.

Example 7-6 Execution Plan

```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%' ;
```

Execution Plan

 Plan hash value: 975837011

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3	189	7 (15)	00:00:01
* 1	HASH JOIN		3	189	7 (15)	00:00:01
* 2	HASH JOIN		3	141	5 (20)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	3	60	2 (0)	00:00:01
* 4	INDEX RANGE SCAN	EMP_NAME_IX	3		1 (0)	00:00:01
5	TABLE ACCESS FULL	JOBS	19	513	2 (0)	00:00:01
6	TABLE ACCESS FULL	DEPARTMENTS	27	432	2 (0)	00:00:01

Predicate Information (identified by operation id):

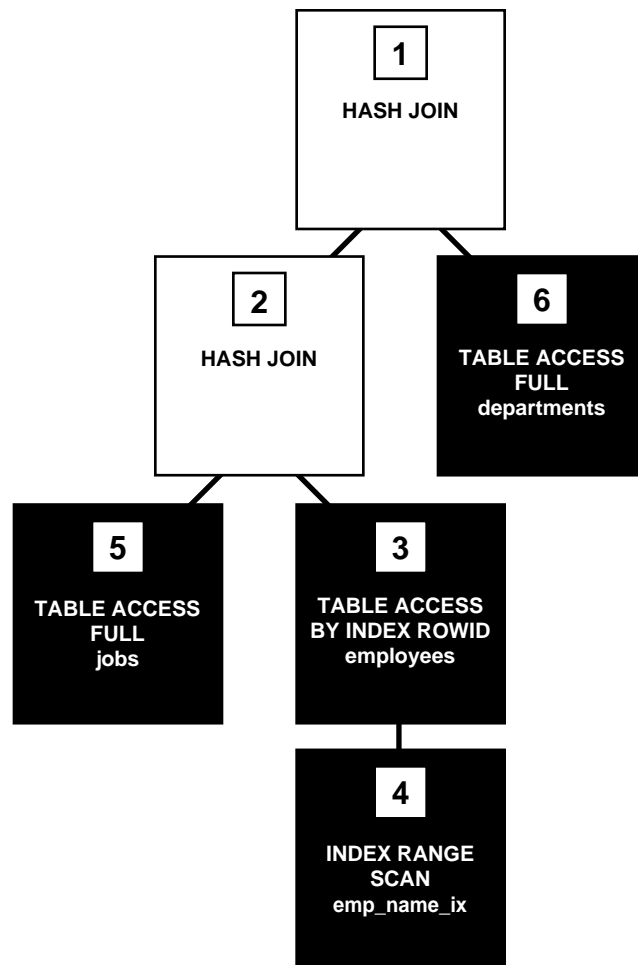
```
-----
1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
2 - access("E"."JOB_ID"="J"."JOB_ID")
4 - access("E"."LAST_NAME" LIKE 'A%')
   filter("E"."LAST_NAME" LIKE 'A%')
```

SQL Execution

During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing.

Figure 7-5 is an **execution tree**, also called a **parse tree**, that shows the flow of row sources from one step to another. In general, the order of the steps in execution is the *reverse* of the order in the plan, so you read the plan from the bottom up. Initial spaces in the Operation column indicate hierarchical relationships. For example, if the name of an operation is preceded by two spaces, then this operation is a child of an operation preceded by one space. Operations preceded by one space are children of the SELECT statement itself.

Figure 7-5 Row Source Tree



In [Figure 7-5](#), each node of the tree acts as a row source, which means that each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input. The SQL engine executes each row source as follows:

- Steps indicated by the black boxes physically retrieve data from an object in the database. These steps are the **access paths**, or techniques for retrieving data from the database.
 - Step 6 uses a **full table scan** to retrieve all rows from the departments table.
 - Step 5 uses a full table scan to retrieve all rows from the jobs table.
 - Step 4 scans the emp_name_ix index in order, looking for each key that begins with the letter A and retrieving the corresponding rowid (see "[Index Range Scan](#)" on page 3-7). For example, the rowid corresponding to Atkinson is AAAPzRAAFAAAABSAAe.
 - Step 3 retrieves from the employees table the rows whose rowids were returned by Step 4. For example, the database uses rowid AAAPzRAAFAAAABSAAe to retrieve the row for Atkinson.
- Steps indicated by the clear boxes operate on row sources.
 - Step 2 performs a **hash join**, accepting row sources from Steps 3 and 5, joining each row from the Step 5 row source to its corresponding row in Step 3, and returning the resulting rows to Step 1.

For example, the row for employee Atkinson is associated with the job name Stock Clerk.

- Step 1 performs another hash join, accepting row sources from Steps 2 and 6, joining each row from the Step 6 source to its corresponding row in Step 2, and returning the result to the client.

For example, the row for employee Atkinson is associated with the department named Shipping.

In some execution plans the steps are iterative and in others sequential. The plan shown in [Example 7–6](#) is iterative because the SQL engine moves from index to table to client and then repeats the steps.

During execution, the database reads the data from disk into memory if the data is not in 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.

See Also: *Oracle Database Performance Tuning Guide* for detailed information about execution plans and the `EXPLAIN PLAN` statement

How Oracle Database Processes DML

Most DML statements have a query component. In a query, execution of a cursor places the results of the query into a set of rows called the **result set**.

Result set rows can be fetched either a row at a time or in groups. In the fetch stage, the database selects rows and, if requested by the query, orders the rows. Each successive fetch retrieves another row of the result until the last row has been fetched.

In general, the database cannot determine for certain the number of rows to be retrieved by a query until the last row is fetched. Oracle Database retrieves the data in response to fetch calls, so that the more rows the database reads, the more work it performs. For some queries the database returns the first row as quickly as possible, whereas for others it creates the entire result set before returning the first row.

Read Consistency

In general, a query retrieves data by using the Oracle Database **read consistency** mechanism. This mechanism, which uses **undo data** to show past versions of data, guarantees that all **data blocks** read by a query are consistent to a single point in time.

For an example of read consistency, suppose a query must read 100 data blocks in a **full table scan**. The query processes the first 10 blocks while DML in a different session modifies block 75. When the first session reaches block 75, it realizes the change and uses undo data to retrieve the old, unmodified version of the data and construct a noncurrent version of block 75 in memory.

See Also: ["Multiversion Read Consistency"](#) on page 9-2

Data Changes

DML statements that must change data use the read consistency mechanism to retrieve only the data that matched the search criteria when the modification began. Afterward, these statements retrieve the data blocks as they exist in their current state and make the required modifications. The database must perform other actions related to the modification of the data such as generating redo and undo data.

See Also: ["Overview of the Online Redo Log"](#) on page 11-12

How Oracle Database Processes DDL

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.

The database process DDL differently because it is a means of defining an object in the data dictionary. Typically, Oracle Database must parse and execute many **recursive SQL** statements to execute a DDL command. Suppose you create a table as follows:

```
CREATE TABLE mytable (mycolumn INTEGER);
```

Typically, the database would run dozens of recursive statements to execute the preceding statement. The recursive SQL would perform actions such as the following:

- Issue a `COMMIT` before executing the `CREATE TABLE` statement
- Verify that user privileges are sufficient to create the table
- Determine which tablespace the table should reside in
- Ensure that the tablespace quota has not been exceeded
- Ensure that no object in the schema has the same name
- Insert rows that define the table into the data dictionary
- Issue a `COMMIT` if the DDL statement succeeded or a `ROLLBACK` if it did not

See Also: *Oracle Database Advanced Application Developer's Guide* to learn about SQL processing for application developers

Server-Side Programming: PL/SQL and Java

[Chapter 7, "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](#)

See Also: [Chapter 7, "SQL"](#)

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:

- "[Client-Side Database Programming](#)" on page 19-5 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 Advanced Application Developer's 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 **schema objects** called **PL/SQL program units**.

PL/SQL program units generally are categorized as follows:

- A **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.
- An **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.

See Also:

- ["Tools for Database Developers"](#) on page 19-1
- *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 **procedure** or a **function**. Procedures and functions are identical except that functions always return a single value to the caller, whereas procedures do not. The term **procedure** in this chapter means **procedure or function**.

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

As explained in ["Introduction to Server-Side Programming"](#) on page 8-1, server-side programming has many advantages over client-side programming. PL/SQL subprograms provide the following advantages:

- 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 (see "[Overview of Database Security](#)" on page 17-1). A **definer's rights procedure** executes with the **privileges** of its owner, not its current user. Thus, you can restrict the database operations 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 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 subprogram created at the schema level with the `CREATE PROCEDURE` or `CREATE FUNCTION` statement is a **standalone stored subprogram**. 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 8-1](#) shows part of a creation statement for the standalone PL/SQL procedure `hire_employees`. The procedure inserts a row into the `employees` table.

Example 8–1 PL/SQL Procedure

```
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:

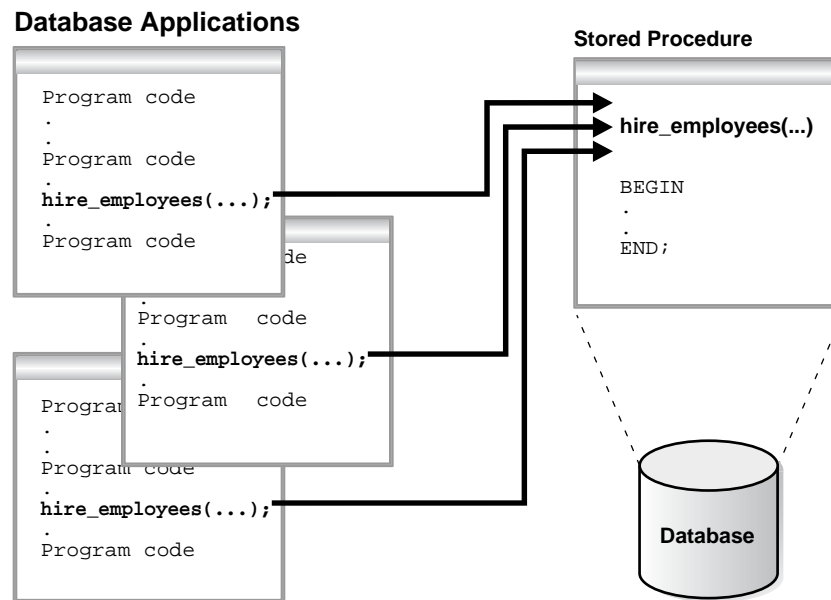
- *Oracle Database 2 Day Developer's Guide* to learn how to create subprograms
- *Oracle Database PL/SQL Language Reference* to learn about the `CREATE PROCEDURE` command

Execution of PL/SQL Subprograms

Users can execute a subprogram interactively by:

- Using an Oracle tool, such as SQL*Plus or SQL Developer (see "[Tools for Database Developers](#)" on page 19-1)
- Calling it explicitly in the code of a database application, such as an Oracle Forms or precompiler application (see "[Client-Side Database Programming](#)" on page 19-5)
- Calling it explicitly in the code of another procedure or trigger

[Figure 8–1](#) shows different database applications calling `hire_employees`.

Figure 8–1 Calling a PL/SQL 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 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 the following advantages:

- 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. As a result, 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 **body** defines all constructs (public and private) of the package.

[Example 8-1](#) 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.

Example 8-2 PL/SQL Package

```
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.

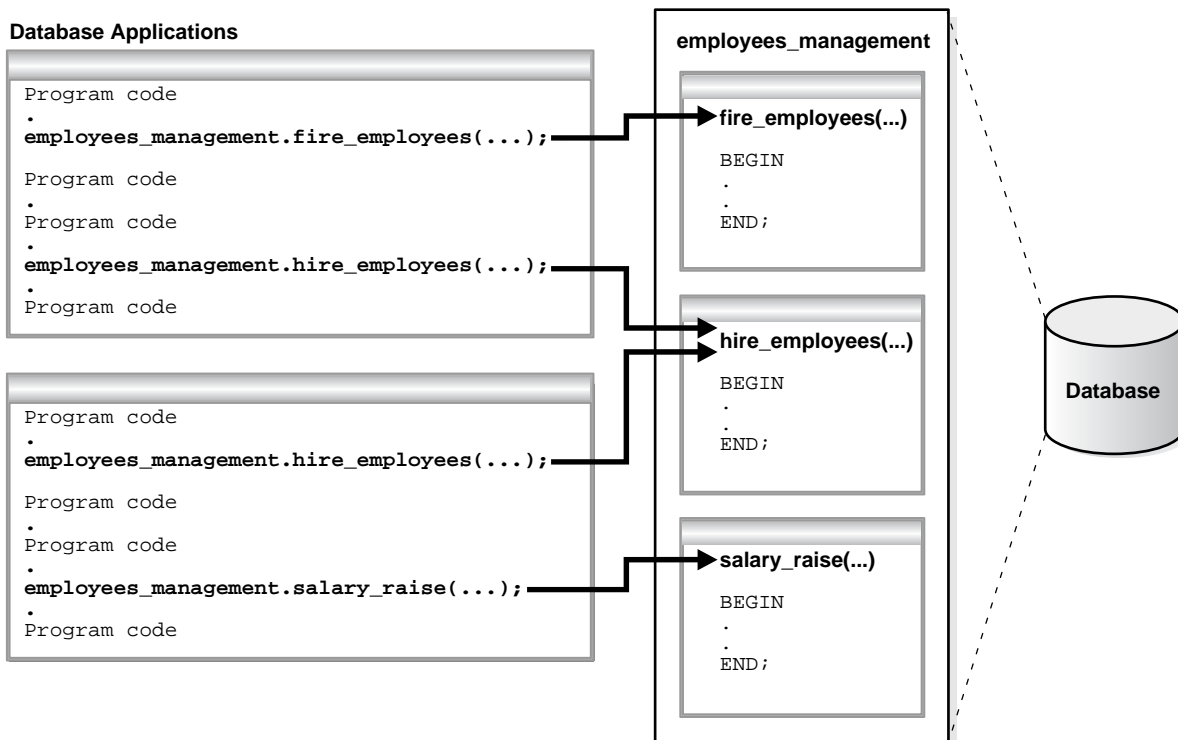
The `CREATE PACKAGE BODY` command defines objects declared in the specification. The package body must be created in the same schema as the package. After creating the package, you can develop applications that call any of these public procedures or functions or raise any of the public exceptions of the package.

See Also: *Oracle Database PL/SQL Language Reference* to learn about the `CREATE PACKAGE` command

Execution of PL/SQL Package Subprograms

You can reference package contents from database triggers, stored subprograms, 3GL application programs, and Oracle tools. [Figure 8–2](#) shows database applications invoking procedures and functions in the `employees_management` package.

Figure 8–2 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 Advanced Application Developer's Guide* to learn how to code PL/SQL packages

PL/SQL Anonymous Blocks

An **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 8–1](#) summarizes the differences between the two types of program units.

Table 8–1 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 Advanced Application Developer's 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 Language 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 these techniques in database applications, PL/SQL provides the data types `TABLE` and `VARRAY`, which enable you to declare associative arrays, nested tables, and variable-size arrays.

Collections

A **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 **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* for detailed information on using collections and records

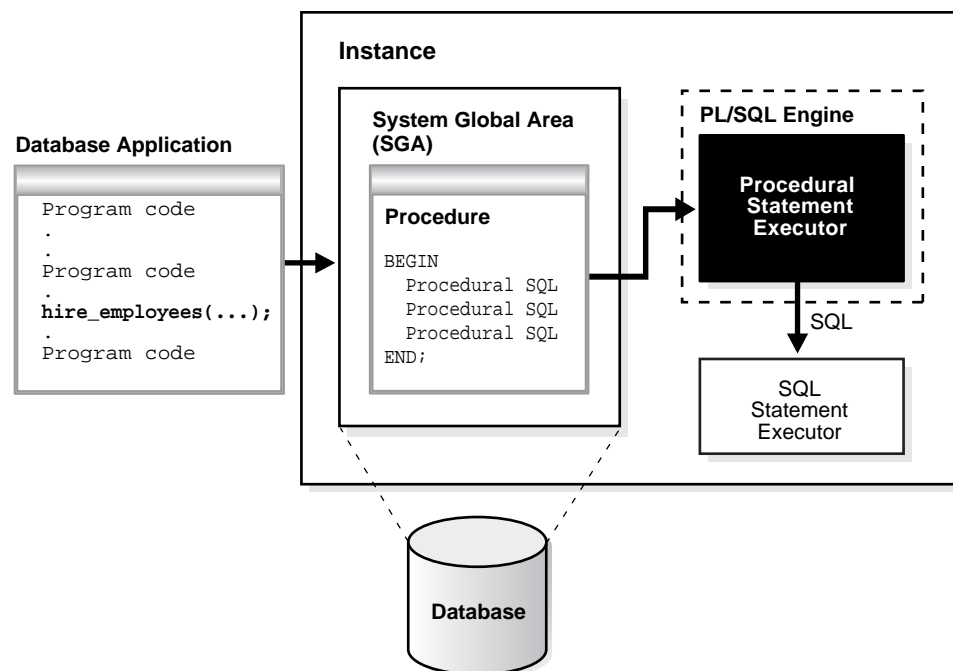
How PL/SQL Runs

PL/SQL supports both **native execution** and **interpreted execution**. In interpreted execution, PL/SQL source code is compiled into a so-called bytecode representation, which is run by a portable virtual computer implemented as part of Oracle Database. In native execution, which offers the best performance on computationally intensive program units, the source code of PL/SQL program units is compiled directly to object code for the given platform. This object code is linked into Oracle Database.

The **PL/SQL engine** is 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. While many Oracle products have PL/SQL components, this section specifically covers the program units that can be stored in Oracle Database and processed using Oracle Database PL/SQL engine. The PL/SQL capabilities of each Oracle tool are described in the documentation for this tool.

Figure 8-3 illustrates the PL/SQL engine contained in Oracle Database.

Figure 8-3 The PL/SQL Engine and Oracle Database



The program unit is stored in a database. When an application calls a stored procedure, the database loads the compiled program unit into the **shared pool** in the **system global area (SGA)** (see "Shared Pool" on page 14-15). 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:

- *Oracle Database PL/SQL Language Reference* to learn about PL/SQL architecture
- *Oracle Database Advanced Application Developer's 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

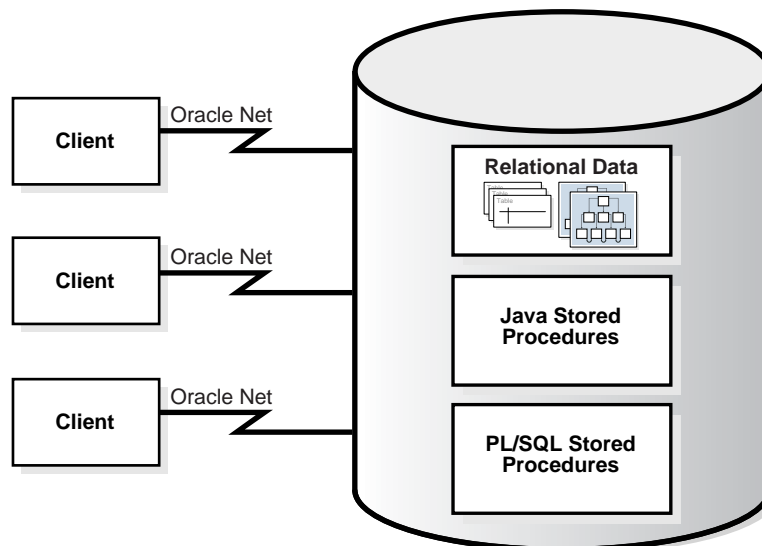
Note: This chapter assumes that you have some familiarity with the Java language.

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.

Figure 8-4 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 8-4 Two-Tier Client/Server Configuration



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)

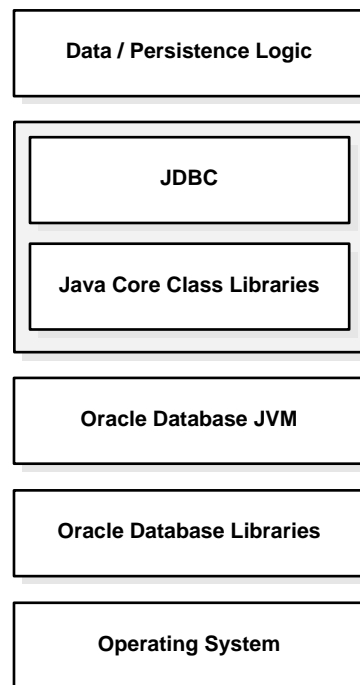
A JVM is a virtual processor that runs compiled Java code. Java source code compiles to low-level machine instructions, known as **bytecodes**, that are platform independent. The Java bytecodes are interpreted through the JVM into platform-dependent actions.

Overview of Oracle JVM

Oracle JVM is a complete, Java2-compliant environment for running pure Java applications. It is compatible with the JLS and the JVM specifications. It 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.

[Figure 8–5](#) 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 8–5 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 you want the JVM to interpret 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

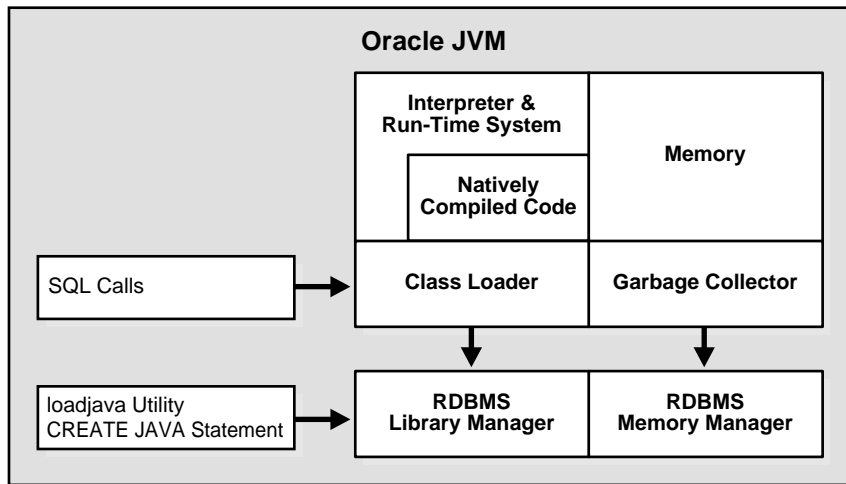
Main Components of Oracle JVM

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`.

Figure 8–6 shows the main components of Oracle JVM.

Figure 8–6 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. As a result, 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. 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. Both approaches are available on the client and server. As a result, you can deploy applications on the client and server without modifying the code.

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.

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.

Driver	Description
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:

- "ODBC and JDBC" on page 19-8
- *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 SQLJ in stored procedures, triggers, and methods within the Oracle Database environment. In addition, you can combine SQLJ programs with JDBC.

SQLJ provides a simple, but powerful, way to develop client-side and middle-tier applications that access databases from Java (see "SQLJ" on page 19-6). 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);
```

Because Oracle Database provides a complete Java environment, you cannot compile SQLJ programs on a client that will run on the database. Instead, you can compile them directly on the server.

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 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.

See Also:

- ["Overview of SQL Statements"](#) on page 7-3 to learn about DML and DDL
- ["Overview of Instance Startup and Shutdown"](#) on page 13-5

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](#) (see ["Introduction to Data Integrity"](#) on page 5-1).

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: *Oracle Database 2 Day Developer's Guide* and *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 with respect to other triggers 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.

Suppose that you create the `orders` and `lineitems` tables as follows:

```
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 `orders` table contains a row for each unique order, whereas the `lineitems` table contains a row for each item in an order. [Example 8-3](#) shows a sample trigger that automatically updates the `orders` table with the number of items in an order.

Example 8-3 *lineitems_trigger*

```
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 [Example 8-3](#), 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 rows in `lineitems` are inserted or updated 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. [Table 8–2](#) illustrates a scenario in which a customer initiates two orders and adds and removes line items from the orders.

Table 8–2 Row-Level Trigger Scenario

SQL Statement	Triggered SQL Statement	Description
<pre>SQL> INSERT INTO orders (order_id) VALUES (78); 1 row created.</pre>		<p>The customer creates an order with ID 78. At this point the customer has no items in the order.</p> <p>Because no action is performed on the <code>lineitems</code> table, the trigger is not invoked.</p>
<pre>SQL> INSERT INTO orders (order_id) VALUES (92); 1 row created.</pre>		<p>The customer creates a separate order with ID 92. At this point the customer has no items in the order.</p> <p>Because no action is performed on the <code>lineitems</code> table, the trigger is not invoked.</p>
<pre>SQL> INSERT INTO lineitems (order_id, seq_no) VALUES (78,1); 1 row created.</pre>	<pre>UPDATE orders SET line_items_count = NVL(NULL,0)+1 WHERE order_id = 78;</pre>	<p>The customer adds an item to order 78.</p> <p>The <code>INSERT</code> invokes the trigger. The triggered statement increases the line item count for order 78 from 0 to 1.</p>
<pre>SQL> INSERT INTO lineitems (order_id, seq_no) VALUES (78,2); 1 row created.</pre>	<pre>UPDATE orders SET line_items_count = NVL(1,0)+1 WHERE order_id = 78;</pre>	<p>The customer adds an additional item to order 78.</p> <p>The <code>INSERT</code> invokes the trigger. The triggered statement increases the line item count for order 78 from 1 to 2.</p>
<pre>SQL> SELECT * FROM orders; ORDER_ID LINE_ITEMS_COUNT ----- 78 2 92 0</pre>		<p>The customer queries the status of the two orders. Order 78 contains two items. Order 92 contains no items.</p>
<pre>SQL> SELECT * FROM lineitems; ORDER_ID SEQ_NO ----- 78 1 78 2</pre>		<p>The customer queries the status of the line items. Each item is uniquely identified by the order ID and the sequence number.</p>
<pre>SQL> UPDATE lineitems SET order_id = 92; 2 rows updated.</pre>	<pre>UPDATE orders SET line_items_count = NVL(NULL,0)+1 WHERE order_id = 92; UPDATE orders SET line_items_count = NVL(2,0)-1 WHERE order_id = 78; UPDATE orders SET line_items_count = NVL(1,0)+1 WHERE order_id = 92; UPDATE orders SET line_items_count = NVL(1,0)-1 WHERE order_id = 78;</pre>	<p>The customer moves the line items that were in order 78 to order 92.</p> <p>The <code>UPDATE</code> statement changes 2 rows in the <code>lineitems</code> tables, which invokes the trigger once for each row.</p> <p>Each time the trigger is invoked, both <code>IF</code> 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 <code>UPDATE</code> statements are run.</p>

Table 8–2 (Cont.) Row-Level Trigger Scenario

SQL Statement	Triggered SQL Statement	Description
<pre>SQL> SELECT * FROM orders; ORDER_ID LINE_ITEMS_COUNT ----- 78 0 92 2</pre>		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.
<pre>SQL> SELECT * FROM lineitems; ORDER_ID SEQ_NO ----- 92 1 92 2</pre>		The customer queries the status of the line items. Each item is uniquely identified by the order ID and the sequence number.
<pre>SQL> DELETE FROM lineitems; 2 rows deleted.</pre>	<pre>UPDATE orders SET line_items_count = NVL(2,0)-1 WHERE order_id = 92; UPDATE orders SET line_items_count = NVL(1,0)-1 WHERE order_id = 92;</pre>	<p>The customer now removes all line items from all orders.</p> <p>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.</p>
<pre>SQL> SELECT * FROM orders; ORDER_ID LINE_ITEMS_COUNT ----- 78 0 92 0 SQL> SELECT * FROM lineitems; no rows selected</pre>		<p>The customer queries the status of the two orders. Neither order contains line items.</p> <p>The customer also queries the status of the line items. No items exist.</p>

See Also:

- *Oracle Database 2 Day Developer's Guide* and *Oracle Database PL/SQL Language Reference* to learn how to create triggers
- *Oracle Database PL/SQL Language Reference* to learn about the CREATE TRIGGER command

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 has the right to fire a trigger if he or she has the privilege to run the triggering statement. With this exception, the database validates and runs triggers the same way as stored subprograms.

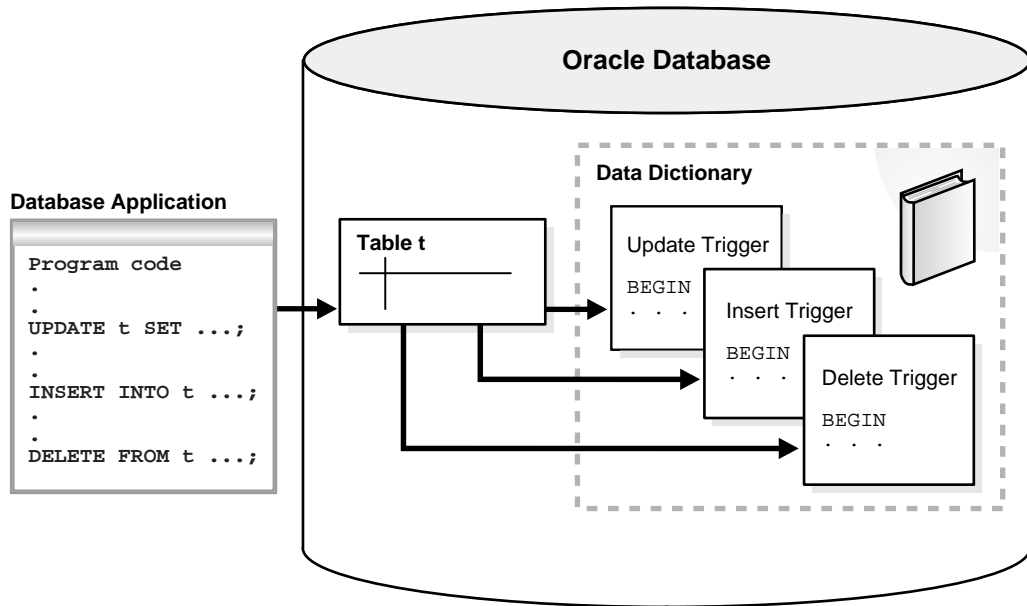
See Also: *Oracle Database PL/SQL Language Reference* to learn more about trigger execution

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 and the source code of the PL/SQL trigger is removed from the [shared pool](#).

Figure 8–7 shows a database application with SQL statements that implicitly invoke PL/SQL triggers. The triggers are stored separately from their associated tables.

Figure 8–7 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 III

Oracle Transaction Management

This part contains the following chapters:

- [Chapter 10, "Transactions"](#)
- [Chapter 9, "Data Concurrency and Consistency"](#)

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** can update the same data. Transactions executing simultaneously must produce meaningful and consistent results. Therefore, a multiuser database must provide the following:

- **Data concurrency**, which ensures that users can access data at the same time
- **Data consistency**, which ensures that each user sees a consistent view of the data, 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.

See Also: [Chapter 5, "Data Integrity"](#) and [Chapter 10, "Transactions"](#)

Multiversion Read Consistency

In Oracle Database, **multiversioning** is the ability to simultaneously materialize multiple versions of data. Oracle Database maintains **multiversion read consistency**, which means that database queries have the following characteristics:

- Read-consistent queries

The data returned by a query is committed and consistent with respect to a single point in time.

Important: Oracle Database *never* permits **dirty reads**, which occur 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 (see "[Summary of Locking Behavior](#)" on page 9-12).

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 with respect to 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 Advanced Application Developer's 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, which 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 this goal through **undo data**.

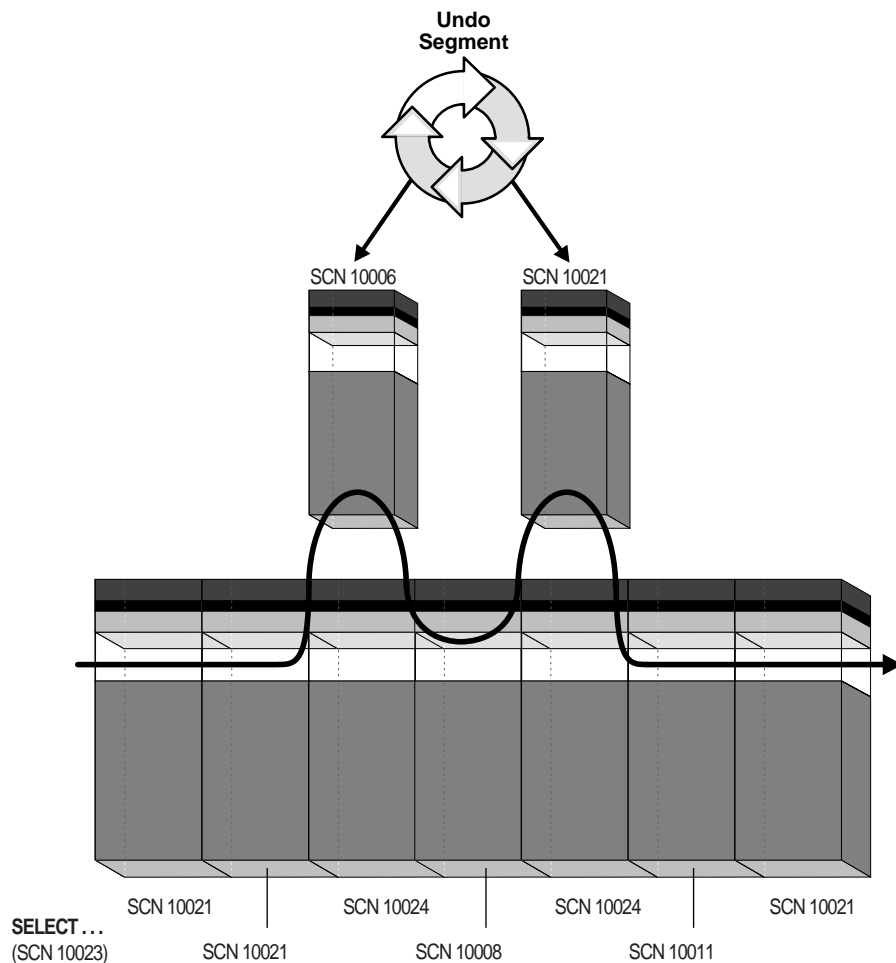
Whenever a user modifies data, Oracle Database creates undo entries, which it writes to undo **segments** ("Undo Segments" on page 12-24). The undo segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Thus, multiple versions of the same data, 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.

See Also:

- "[Internal LOBs](#)" on page 19-12 to learn about read consistency mechanisms for LOBs
- *Oracle Database 2 Day + Real Application Clusters Guide* to learn about Cache Fusion

Read Consistency: Example [Figure 9–1](#) shows a query that uses undo data to provide statement-level read consistency in the read committed isolation level.

Figure 9–1 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 a 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 9–1](#), this SCN is 10023. The query only sees committed data with respect to SCN 10023.

In [Figure 9–1](#), 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 9–1](#), 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"](#) on page 14-9 and ["System Change Numbers \(SCNs\)"](#) on page 10-5

Read Consistency and Transaction Tables The database uses a **transaction table**, also called an **interested transaction list (ITL)**, to determine if a transaction was uncommitted when the database began modifying the block. The **block header** of every segment block contains a transaction table.

Entries in the transaction table describe which transactions have rows locked and which rows in the block contain committed and uncommitted changes. The transaction table points to 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

Locking Mechanisms

In general, multiuser databases use some form of data locking to solve the problems associated with data concurrency, consistency, and integrity. **Locks** are mechanisms that prevent destructive interaction between transactions accessing the same resource.

See Also: "[Overview of the Oracle Database Locking Mechanism](#)" on page 9-11

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 9-1](#) shows the levels.

Table 9–1 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"](#) on page 9-6 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

[Table 9–1](#) summarizes the ANSI standard for transaction isolation levels. The standard 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](#)

See Also:

- *Oracle Database Advanced Application Developer's 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**, which is the default, every query executed by a transaction sees only data committed before the query—not the transaction—began. This level of isolation 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

A consistent result set is provided 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 and Implicit Queries"](#) on page 7-7

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, 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.

[Table 9–2](#) shows how transaction 1, which can be either serializable or read committed, interacts with read committed transaction 2. [Table 9–2](#) shows a classic situation known as a **lost update** (see ["Use of Locks"](#) on page 9-12). 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 9–2 *Conflicting Writes and Lost Updates in a READ COMMITTED Transaction*

Session 1	Session 2	Explanation
<pre>SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</pre> <pre> LAST_NAME SALARY ----- Banda 6200 Greene 9500 </pre>		Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.
<pre>SQL> UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';</pre>		Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is READ COMMITTED.
	<pre>SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;</pre>	Session 2 begins transaction 2 and sets the isolation level explicitly to READ COMMITTED.
	<pre>SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</pre> <pre> LAST_NAME SALARY ----- Banda 6200 Greene 9500 </pre>	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 9–2 (Cont.) Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

Session 1	Session 2	Explanation
	SQL> UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';	Transaction 2 updates the salary for Greene successfully because transaction 1 locked only the Banda row (see "Row Locks (TX)" on page 9-18).
SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');		Transaction 1 inserts a row for employee Hintz, but does not commit.
	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz'); LAST_NAME SALARY ----- Banda 6200 Greene 9900	Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. 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.
	SQL> UPDATE employees SET salary = 6300 WHERE last_name = 'Banda'; -- 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;		Transaction 1 commits its work, ending the transaction.
	1 row updated. SQL>	The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary for Banda.
	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz'); LAST_NAME SALARY ----- Banda 6300 Greene 9900 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.
	COMMIT;	Transaction 2 commits its work, ending the transaction.
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz'); LAST_NAME SALARY ----- Banda 6300 Greene 9900 Hintz		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."

Serializable Isolation Level

In the **serialization 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

Table 9–3 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 9–3 Read Consistency and Serialized Access Problems in Serializable Transactions

Session 1	Session 2	Explanation
<pre>SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');</pre> <pre>LAST_NAME SALARY ----- Banda 6200 Greene 9500</pre>		Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.
<pre>SQL> UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';</pre>		Session 1 begins transaction 1 by updating the Banda salary. The default isolation level for is READ COMMITTED.
	<pre>SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</pre>	Session 2 begins transaction 2 and sets it to the SERIALIZABLE isolation level.
	<pre>SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda','Greene','Hintz');</pre> <pre>LAST_NAME SALARY ----- Banda 6200 Greene 9500</pre>	Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda <i>before</i> the uncommitted update made by transaction 1.
	<pre>SQL> UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';</pre>	Transaction 2 updates the Greene salary successfully because only the Banda row is locked.

Table 9–3 (Cont.) Read Consistency and Serialized Access Problems in Serializable Transactions

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');		Transaction 1 inserts a row for employee Hintz.
SQL> COMMIT;		Transaction 1 commits its work, ending the transaction.
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz'); LAST_NAME SALARY ----- Banda 7000 Greene 9500 Hintz	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz'); LAST_NAME SALARY ----- Banda 6200 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. 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 <i>not</i> visible to transaction 2. Transaction 2 sees its own update to the Banda salary.
	COMMIT;	Transaction 2 commits its work, ending the transaction.
SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz'); LAST_NAME SALARY ----- Banda 7000 Greene 9900 Hintz	SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', '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';		Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is READ COMMITTED.
	SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Session 2 begins transaction 4 and sets it to the SERIALIZABLE isolation level.
	SQL> UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz'; -- prompt does not return	Transaction 4 attempts to update the salary for Hintz, but is blocked because transaction 3 locked the Hintz row (see "Row Locks (TX)" on page 9-18). Transaction 4 queues behind transaction 3.
SQL> COMMIT;		Transaction 3 commits its update of the Hintz salary, ending the transaction.
	UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz' * ERROR at line 1: ORA-08177: can't serialize access for 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 <i>after</i> transaction 4 began.
	SQL> ROLLBACK;	Session 2 rolls back transaction 4, which ends the transaction.
	SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Session 2 begins transaction 5 and sets it to the SERIALIZABLE isolation level.

Table 9–3 (Cont.) Read Consistency and Serialized Access Problems in Serializable Transactions

Session 1	Session 2	Explanation
	<pre>SQL> SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</pre> <pre> LAST_NAME SALARY ----- Banda 7100 Greene 9500 Hintz 7100 </pre>	Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible.
	<pre>SQL> UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz';</pre> <pre> 1 row updated. </pre>	<p>Transaction 5 updates the Hintz salary to a different value. Because the Hintz update made by transaction 3 committed <i>before</i> the start of transaction 5, the serialized access problem is avoided.</p> <p>Note: If a different transaction updated and committed the Hintz row after transaction 5 began, then the serialized access problem would occur again.</p>
	<pre>SQL> COMMIT;</pre>	Session 2 commits the update without any problems, ending the transaction.

See Also: ["Overview of Transaction Control"](#) on page 10-6

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. Thus, 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"](#) on page 12-24
- *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**, which are interactions that 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. 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 circumstances¹ 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.
- 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** 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.

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.

- 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 is performed 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** database problem in which one user overwrites changes made by another user, effectively losing the update by the second user (Table 9-2 on page 9-7 shows an example of a lost update).

Table 9-4 shows the sequence of events when two sessions attempt to modify the same row in the employees table at roughly the same time.

Table 9-4 Row Locking Example

Time	Session 1	Session 2	Explanation
t0	<pre>SELECT employee_id, email, phone_number FROM hr.employees WHERE last_name = 'Himuro'; EMPLOYEE_ID EMAIL PHONE_NUMBER ----- 118 GHIMURO 515.127.4565</pre>		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		<pre>SELECT employee_id, email, phone_number FROM hr.employees WHERE last_name = 'Himuro'; EMPLOYEE_ID EMAIL PHONE_NUMBER ----- 118 GHIMURO 515.127.4565</pre>	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	<pre>UPDATE hr.employees SET phone_number='515.555.1234' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.127.4565'; 1 row updated.</pre>		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.

Table 9–4 (Cont.) Row Locking Example

Time	Session 1	Session 2	Explanation
t3		<pre>UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.127.4565'; -- SQL*Plus does not show -- a row updated message or -- return the prompt.</pre>	<p>In session 2, the hr2 user attempts to update the same row, but is blocked because hr1 is currently processing the row.</p> <p>The attempted update by hr2 occurs almost simultaneously with the hr1 update.</p>
t4	<pre>COMMIT; Commit complete.</pre>		<p>In session 1, the hr1 user commits the transaction.</p> <p>The commit makes the change for Himuro permanent and unblocks session 2, which has been waiting.</p>
t5		0 rows updated.	<p>In session 2, the hr2 user discovers that the GHIMURO row was modified in such a way that it no longer matches its predicate.</p> <p>Because the predicates do not match, session 2 updates no records.</p>
t6	<pre>UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.555.1234'; 1 row updated.</pre>		<p>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.</p>
t7		<pre>SELECT employee_id, email, phone_number FROM hr.employees WHERE last_name = 'Himuro'; EMPLOYEE_ID EMAIL PHONE_NUMBER ----- 118 GHIMURO 515.555.1234</pre>	<p>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.</p>
t8		<pre>UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.555.1234'; -- SQL*Plus does not show -- a row updated message or -- return the prompt.</pre>	<p>In session 2, the hr2 user attempts to update the same row, but is blocked because hr1 is currently processing the row.</p>
t9	<pre>ROLLBACK; Rollback complete.</pre>		<p>In session 1, the hr1 user rolls back the transaction, which ends it.</p>
t10		1 row updated.	<p>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.</p>
t11		<pre>COMMIT; Commit complete.</pre>	<p>Session 2 commits the update, ending the transaction.</p>

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, holding share locks to prevent concurrent access by a writer who needs an exclusive lock. Several 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 user 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 likelihood 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. In most cases, 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, as explained in "[Overview of User-Defined Locks](#)" on page 9-27, 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 [Table 10–2](#) on page 10-9 for an example).

See Also: "[Rollback to Savepoint](#)" on page 10-8

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 signalled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

[Table 9–5](#) illustrates two transactions in a deadlock.

Table 9–5 *Deadlock Example*

Time	Session 1	Session 2	Explanation
t0	SQL> UPDATE employees SET salary = salary*1.1 WHERE employee_id = 100; 1 row updated.	SQL> UPDATE employees SET salary = salary*1.1 WHERE employee_id = 200; 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.

Table 9–5 (Cont.) Deadlock Example

Time	Session 1	Session 2	Explanation
t1	SQL> UPDATE employees SET salary = salary*1.1 WHERE employee_id = 200; -- prompt does not return	SQL> UPDATE employees salary = salary*1.1 WHERE employee_id = 100; -- 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. 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.
t2	UPDATE employees * ERROR at line 1: ORA-00060: deadlock detected while waiting for resource 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. Note: Only one session in the deadlock actually gets the deadlock error, but either session could get the error.
t3	SQL> COMMIT; 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. SQL>	The update at t1 in transaction 2, which was being blocked by transaction 1, is executed. The prompt is returned.
t5		SQL> COMMIT; 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"](#) on page 9-26
- *Oracle Database Advanced Application Developer's 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 following categories.

Lock	Description
DML Locks	Protect data. For example, table locks lock entire tables, while row locks lock selected rows. See "DML Locks" on page 9-18.

Lock	Description
DDL Locks	Protect the structure of schema objects—for example, the dictionary definitions of tables and views. See " DDL Locks " on page 9-24.
System Locks	Protect internal database structures such as data files. Latches, mutexes, and internal locks are entirely automatic. See " System Locks " on page 9-25.

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).

See Also: ["Oracle Enterprise Manager"](#) on page 18-2

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 9–2](#) 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 9–2 Row and Table Locks

Table EMPLOYEES 

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	MANAGER_ID	DEPARTMENT_ID
 100	King	SKING	17-JUN-87	AD_PRES		90
101	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	100	90
102	De Hann	LDEHANN	13-JAN-93	AD_VP	100	90
103	Hunold	AHUNOLD	03-JAN-90	IT_PROG	102	60

-  Table lock acquired
-  Exclusive row lock (TX) acquired
-  Row being updated

Row Locks and Concurrency Table 9–6 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 9–6 Data Concurrency Example

Time	Session 1	Session 2	Session 3	Explanation																		
t0	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <table border="1"> <thead> <tr> <th>EMPLOYEE_ID</th> <th>SALARY</th> </tr> </thead> <tbody> <tr> <td>100</td> <td>512</td> </tr> <tr> <td>101</td> <td>600</td> </tr> </tbody> </table>	EMPLOYEE_ID	SALARY	100	512	101	600	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <table border="1"> <thead> <tr> <th>EMPLOYEE_ID</th> <th>SALARY</th> </tr> </thead> <tbody> <tr> <td>100</td> <td>512</td> </tr> <tr> <td>101</td> <td>600</td> </tr> </tbody> </table>	EMPLOYEE_ID	SALARY	100	512	101	600	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <table border="1"> <thead> <tr> <th>EMPLOYEE_ID</th> <th>SALARY</th> </tr> </thead> <tbody> <tr> <td>100</td> <td>512</td> </tr> <tr> <td>101</td> <td>600</td> </tr> </tbody> </table>	EMPLOYEE_ID	SALARY	100	512	101	600	Three different sessions simultaneously query the ID and salary of employees 100 and 101. The results returned by each query are identical.
EMPLOYEE_ID	SALARY																					
100	512																					
101	600																					
EMPLOYEE_ID	SALARY																					
100	512																					
101	600																					
EMPLOYEE_ID	SALARY																					
100	512																					
101	600																					
t1	<pre>UPDATE hr.employees SET salary=salary+100 WHERE employee_id=100;</pre>			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.																		

Table 9–6 (Cont.) Data Concurrency Example

Time	Session 1	Session 2	Session 3	Explanation
t2	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <pre>EMPLOYEE_ID SALARY ----- ----- 100 612 101 600</pre>	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <pre>EMPLOYEE_ID SALARY ----- ----- 100 512 101 600</pre>	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <pre>EMPLOYEE_ID SALARY ----- ----- 100 512 101 600</pre>	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		<pre>UPDATE hr.employees SET salary=salary+100 WHERE employee_id=101;</pre>		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	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <pre>EMPLOYEE_ID SALARY ----- ----- 100 612 101 600</pre>	<pre>SELECT employee_id, salaryFROM employees WHERE employee_id IN (100, 101);</pre> <pre>EMPLOYEE_ID SALARY ----- ----- 100 512 101 700</pre>	<pre>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</pre> <pre>EMPLOYEE_ID SALARY ----- ----- 100 512 101 600</pre>	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** (see "Overview of Data Blocks" on page 12-6).

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: *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 Advanced Application Developer's Guide* to learn more about table locks

Locks and Foreign Keys

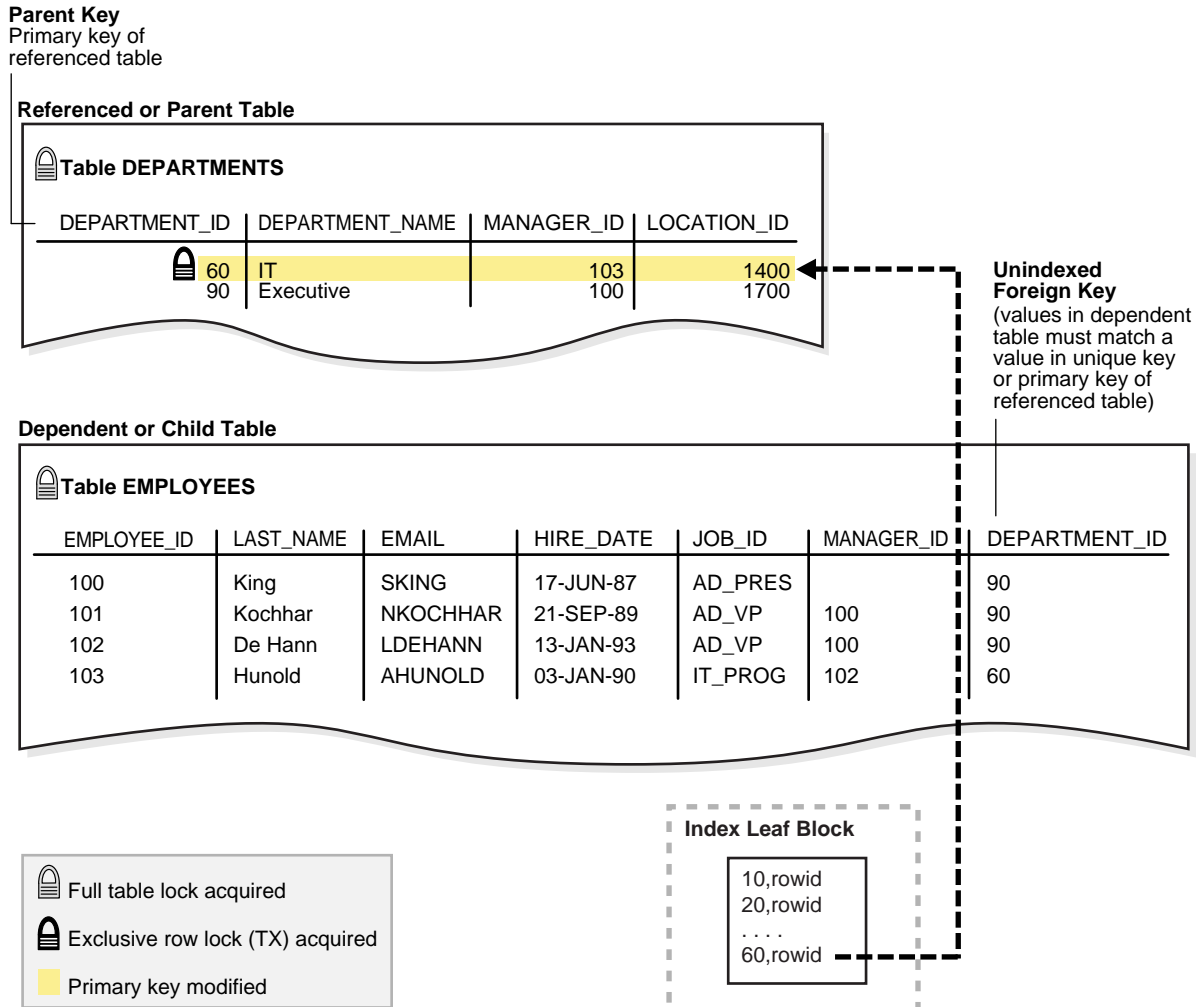
Oracle Database maximizes the concurrency control of parent keys in relation to dependent foreign keys. Locking behavior depends on whether foreign key columns are indexed. If foreign keys are not indexed, then the child table will probably be locked more frequently, deadlocks will occur, and concurrency 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 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. Inserts into the parent table do not acquire table locks on the child table.

Suppose that `hr.departments` table is a parent of `hr.employees`, which contains the unindexed foreign key `department_id`. Figure 9-3 shows a session modifying the primary key attributes of department 60 in the `departments` table.

Figure 9-3 Locking Mechanisms with Unindexed Foreign Key



In Figure 9-3, 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, employee phone numbers cannot be updated. 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 When both of the following conditions are true, the database does *not* acquire a full table lock on the child table:

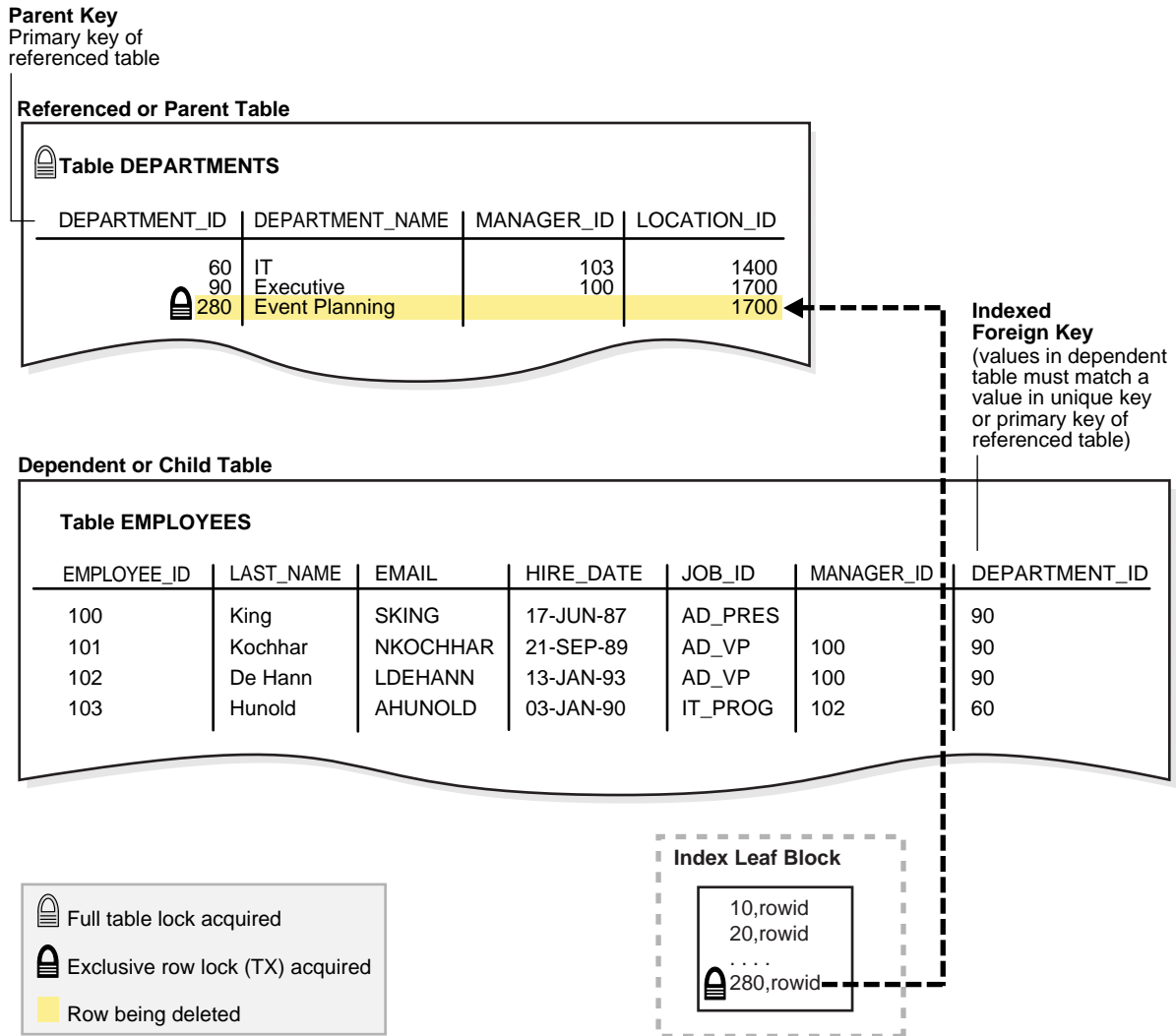
- A foreign key column in the child table is indexed.

- 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 9-4 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" on page 9-21.

Figure 9-4 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"](#) on page 5-6
- ["Overview of Indexes"](#) on page 3-1

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. Most DDL operations, except for those described in ["Share DDL Locks"](#) on page 9-24, 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.

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"](#) on page 14-15

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

Latches are simple, low-level serialization mechanisms that coordinate 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 *DBWn* 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. **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"](#) on page 7-16
- *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" on page 14-19

Overview of Manual Data Locks

Oracle Database performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, you can manually override the Oracle Database default locking mechanisms. 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 ... FOR UPDATE`
- *Oracle Database Advanced Application Developer's 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 Advanced Application Developer's Guide* for more information about Oracle Database Lock Management services
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_LOCK`

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 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. 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 comply with 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

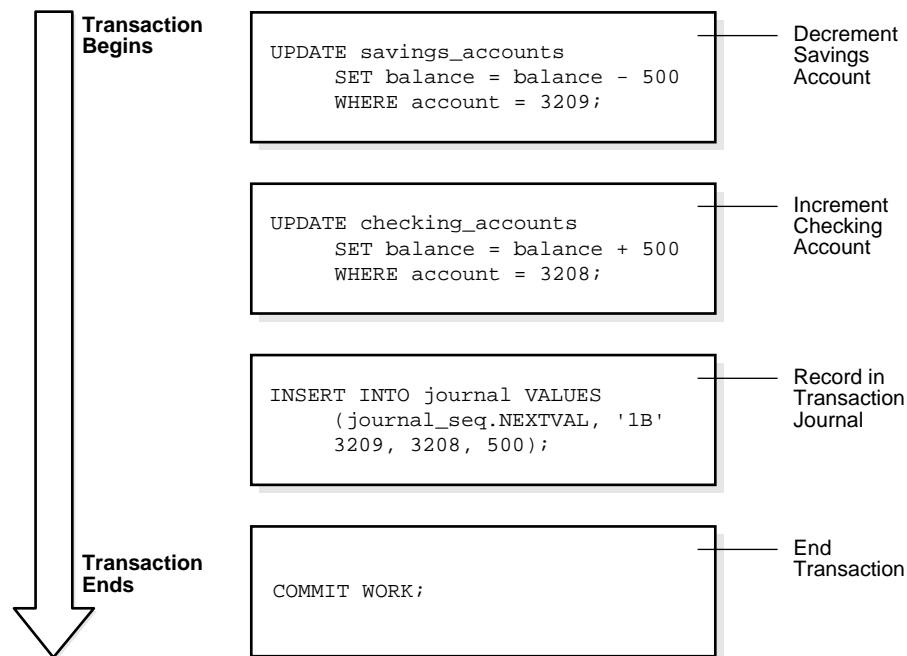
To illustrate the concept of a transaction, consider a banking database. When a customer transfers money from a savings account to a checking account, the transaction must consist of three separate operations:

- 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.

Figure 10–1 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 10–1 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.

See Also: ["Overview of SQL Statements"](#) on page 7-3

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"](#) on page 12-24

End of a Transaction

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. The transaction shown in [Figure 10-1](#) ends with a commit.
- 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:

- ["Tools for Database Administrators"](#) on page 18-2 and ["Tools for Database Developers"](#) on page 19-1
- *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 [Figure 10–1](#) 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"](#) on page 7-16
- ["Locks and Deadlocks"](#) on page 9-16
- ["Overview of Triggers"](#) on page 8-16

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 with respect to the database. Several events may share the same SCN, which means that they occurred at the same time with respect to 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"](#) on page 13-12 and ["Backup and Recovery"](#) on page 18-9

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](#)" on page 7-8:

- 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 10–1](#) illustrates the basic concepts of transaction control.

Table 10–1 Transaction Control

Time	Session	Explanation
t0	<code>COMMIT;</code>	This statement ends any existing transaction in the session.
t1	<code>SET TRANSACTION NAME 'sal_update';</code>	This statement begins a transaction and names it <code>sal_update</code> .
t2	<code>UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';</code>	This statement updates the salary for Banda to 7000.
t3	<code>SAVEPOINT after_banda_sal;</code>	This statement creates a savepoint named <code>after_banda_sal</code> , enabling changes in this transaction to be rolled back to this point.
t4	<code>UPDATE employees SET salary = 12000 WHERE last_name = 'Greene';</code>	This statement updates the salary for Greene to 12000.
t5	<code>SAVEPOINT after_greene_sal;</code>	This statement creates a savepoint named <code>after_greene_sal</code> , enabling changes in this transaction to be rolled back to this point.
t6	<code>ROLLBACK TO SAVEPOINT after_banda_sal;</code>	This statement rolls back the transaction to t3, undoing the update to Greene's salary at t4. The <code>sal_update</code> transaction has <i>not</i> ended.
t7	<code>UPDATE employees SET salary = 11000 WHERE last_name = 'Greene';</code>	This statement updates the salary for Greene to 11000 in transaction <code>sal_update</code> .
t8	<code>ROLLBACK;</code>	This statement rolls back all changes in transaction <code>sal_update</code> , ending the transaction.
t9	<code>SET TRANSACTION NAME 'sal_update2';</code>	This statement begins a new transaction in the session and names it <code>sal_update2</code> .

Table 10–1 (Cont.) Transaction Control

Time	Session	Explanation
t10	UPDATE employees SET salary = 7050 WHERE last_name = 'Banda';	This statement updates the salary for Banda to 7050.
t11	UPDATE employees SET salary = 10950 WHERE last_name = 'Greene';	This statement updates the salary for Greene to 10950.
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.

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 10–1](#) on page 10-6, 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 Enterprise Manager](#)" on page 18-2
- *Oracle Database Reference* to learn about `V$TRANSACTION`
- *Oracle Database SQL Language Reference* to learn about `SET TRANSACTION`

Active Transactions

An **active transaction** has started but not yet committed or rolled back. In [Table 10–1](#) on page 10-6, 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 follows:

- Oracle Database has generated **undo data** information in the **system global area (SGA)**.
The undo data contains the old data values changed by the SQL statements of the transaction. See "Read Consistency in the Read Committed Isolation Level" on page 9-7.
- 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. See "Redo Log Buffer" on page 14-14.
- 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 (DBWn)**. The disk write can happen before or after the commit. See "Database Buffer Cache" on page 14-9.
- 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. See "Summary of Locking Behavior" on page 9-12.

Savepoints

A **savepoint** is a user-declared intermediate marker within the context of a transaction. Internally, this 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 10-1](#) on page 10-6 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. When a transaction is rolled back to a savepoint, as when the `ROLLBACK TO SAVEPOINT after_banda_sal` is run in [Table 10-1](#) on page 10-6, the following occurs:

1. Oracle Database rolls back only the statements run after the savepoint.
In [Table 10-1](#) on page 10-6, 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 10-1](#) on page 10-6, 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 previous to 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 [Table 10–2](#), 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 10–2 Rollback to Savepoint Example

Time	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. At this point, session 2 is <i>still</i> blocked by session 1 because session 2 enqueues on the session 1 <i>transaction</i> , 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"](#) on page 9-16 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 10-1](#) on page 10-6, 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 10-1](#) on page 10-6, 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 10-1](#) on page 10-6, 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"](#) on page 12-24

Committing Transactions

A **commit** ends the current transaction and makes permanent all changes performed in the transaction. In [Table 10-1](#) on page 10-6, 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:

- A **system change number (SCN)** is generated 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. See ["Serializable Isolation Level"](#) on page 9-8.

- The **log writer (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 10-1](#) on page 10-6, 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 from the blocks. Ideally, the `COMMIT` cleans out the blocks so that a subsequent `SELECT` does not have to perform this task.

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:

- *Oracle Database PL/SQL Language Reference* for more information on asynchronous commit
- "[Locking Mechanisms](#)" on page 9-5
- "[Overview of Background Processes](#)" on page 15-7 for more information about LGWR

Overview of Autonomous Transactions

An **autonomous transaction** is an independent transaction that can be called from another transaction, called 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:

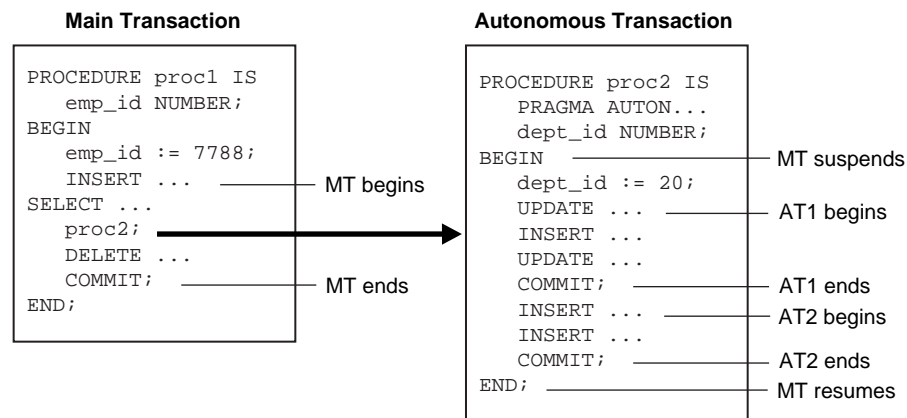
- The autonomous transaction does not see uncommitted changes made by the main transaction and does not share locks or resources with the main transaction.
- Changes in an autonomous transaction are visible to other transactions upon commit of the autonomous transactions. Thus, users can access the updated 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.

Figure 10–2 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 10–2 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 10–2, 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 Advanced Application Developer’s Guide* and *Oracle Database PL/SQL Language Reference* to learn how to use autonomous transactions

Overview of Distributed Transactions

A **distributed database** is a set of databases in a distributed system that can appear to applications as a single data source. 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 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.

See Also: *Oracle Database Administrator's Guide*

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

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 \(RECO\)" on page 15-11](#)
- *Oracle Database Administrator's Guide* to learn how to manage in-doubt transactions

Part IV

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:

- [Chapter 11, "Physical Storage Structures"](#)
- [Chapter 12, "Logical Storage Structures"](#)

Physical Storage Structures

This chapter describes the primary physical database structures of an Oracle database. Physical structures 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. 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.

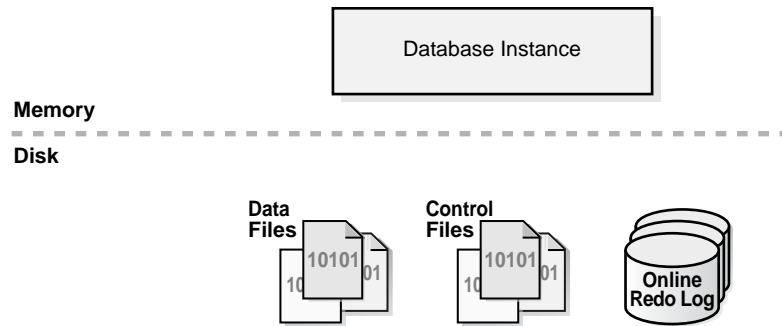
An **Oracle database** is a set of files that store Oracle data in persistent disk storage. This section discusses the database files generated when you issue a `CREATE DATABASE` statement:

- Data files and temp files
 - A **data file** is a physical file on disk 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 data is written to these files in an Oracle proprietary format that cannot be read by other programs.
- Control files
 - A **control file** is a root file that tracks the physical components of the database.
- Online redo log files

The **online redo log** is a set of files containing records of changes made to data.

A database **instance** is a set of memory structures that manage database files.

[Figure 11-1](#) shows the relationship between the instance and the files that it manages.

Figure 11–1 Database Instance and Database 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. The most common mechanisms include:

- Oracle Automatic Storage Management (Oracle ASM)

Oracle ASM includes a file system designed exclusively for use by Oracle Database. "[Oracle Automatic Storage Management \(Oracle ASM\)](#)" on page 11-3 describes Oracle ASM.

- Operating system file system

Most Oracle databases store files in a **file system**, which is a data structure built inside a contiguous disk address space. All operating systems have **file managers** that allocate and deallocate disk space into files within a file system.

A file system enables disk space to be allocated to many files. Each file has a name and is made to appear 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 be combined into a single contiguous address space that appears as one disk to higher layers of software.

- Raw device

Raw devices are disk partitions or logical volumes not formatted with a file system. The primary benefit of raw devices is the ability to perform **direct I/O** and to write larger buffers. In direct I/O, applications write to and read from the storage device directly, bypassing the operating system buffer cache.

Note: Many file systems now support direct I/O for databases and other applications that manage their own caches. Historically, raw devices were the only means of implementing direct I/O.

- Cluster file system

A **cluster file system** is software that enables multiple computers to share file storage while maintaining consistent space allocation and file content. 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.

A database employs a combination of the preceding storage mechanisms. For example, a database could store the control files and online redo log files in a traditional file system, some user data files on raw partitions, the remaining data files in Oracle ASM, and archived the redo log files to a cluster file system.

See Also:

- *Oracle Database 2 Day DBA* to learn how to view database storage structures with Oracle Enterprise Manager (Enterprise Manager)
- *Oracle Database Administrator's Guide* to view database storage structures by querying database views

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**, which is a collection of disks that Oracle ASM manages as a unit. Within a disk group, Oracle ASM exposes a file system interface for database files.

Figure 11–2 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.

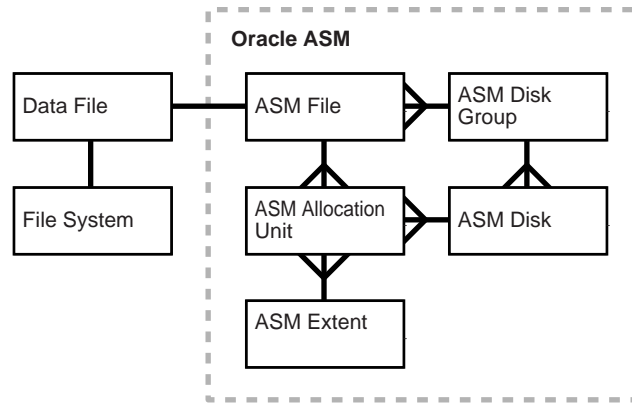
Figure 11–2 Oracle ASM Components

Figure 11–2 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. The performance is comparable to the performance of raw devices.

- 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 fully qualified 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 raw disks and third-party file systems. This capability simplifies the integration of Oracle ASM into pre-existing environments.

- Oracle ASM Extents

An **Oracle ASM extent** is the raw storage used to hold the contents 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 **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:

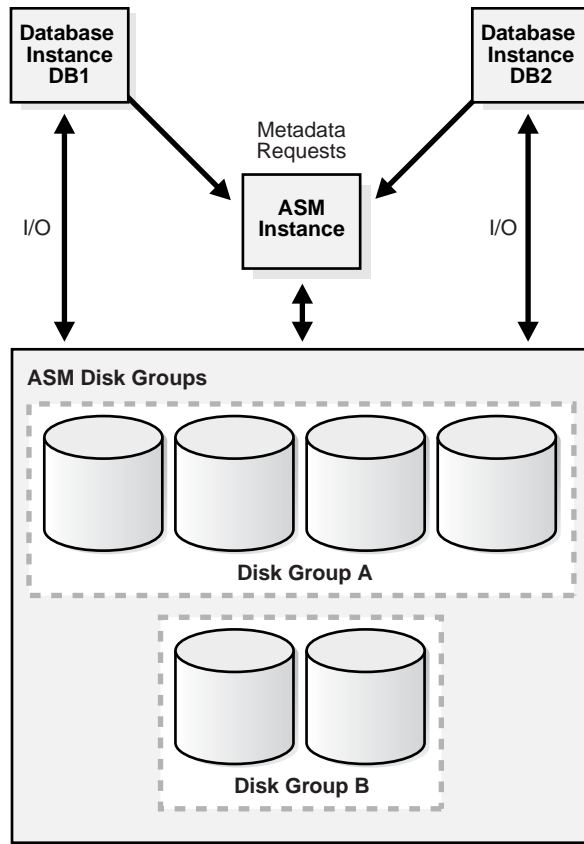
- *Oracle Database 2 Day DBA* to learn how to administer Oracle ASM disks with Oracle Enterprise Manager (Enterprise Manager)
- *Oracle Automatic Storage Management Administrator's Guide* to learn more about Oracle ASM

Oracle ASM Instances

An **Oracle ASM instance** is a special Oracle instance that manages Oracle ASM disks. Both the ASM and the database instances require shared access to the disks in an ASM disk group. ASM instances manage the metadata of the disk group and provide file layout information to the database instances. Database instances direct I/O to ASM disks without going through an ASM instance.

An ASM instance is built on the same technology as a database instance. For example, an ASM instance has a **system global area (SGA)** and background processes that are similar to those of a database instance. However, an ASM instance cannot mount a database and performs fewer tasks than a database instance.

[Figure 11-3](#) shows a single-node configuration with one Oracle ASM instance and two database instances, each associated with a different single-instance database. The ASM instance manages the metadata and provides space allocation for the ASM files storing the data for the two databases. One ASM disk group has four ASM disks and the other has two disks. Both database instances can access the disk groups.

Figure 11–3 Oracle ASM Instance and Database Instances**See Also:**

- *Oracle Database 2 Day DBA* to learn how to administer Oracle ASM disks with Oracle Enterprise Manager (Enterprise Manager)
- *Oracle Automatic Storage Management Administrator's Guide* to learn more about Oracle ASM

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. In this way, 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 (see ["Overview of Diagnostic Files"](#) on page 13-18).

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: *Oracle Database Administrator's Guide* to learn how to use Oracle Managed Files

Overview of Data Files

At the operating system level, Oracle Database stores database data in **data files**. Every database must have at least one data file.

Use of Data Files

Part I, "Oracle Relational Data Structures" explains the logical structures in which users store data, the most important of which are tables. Each nonpartitioned **schema object** and each partition of an object is stored in its own **segment**.

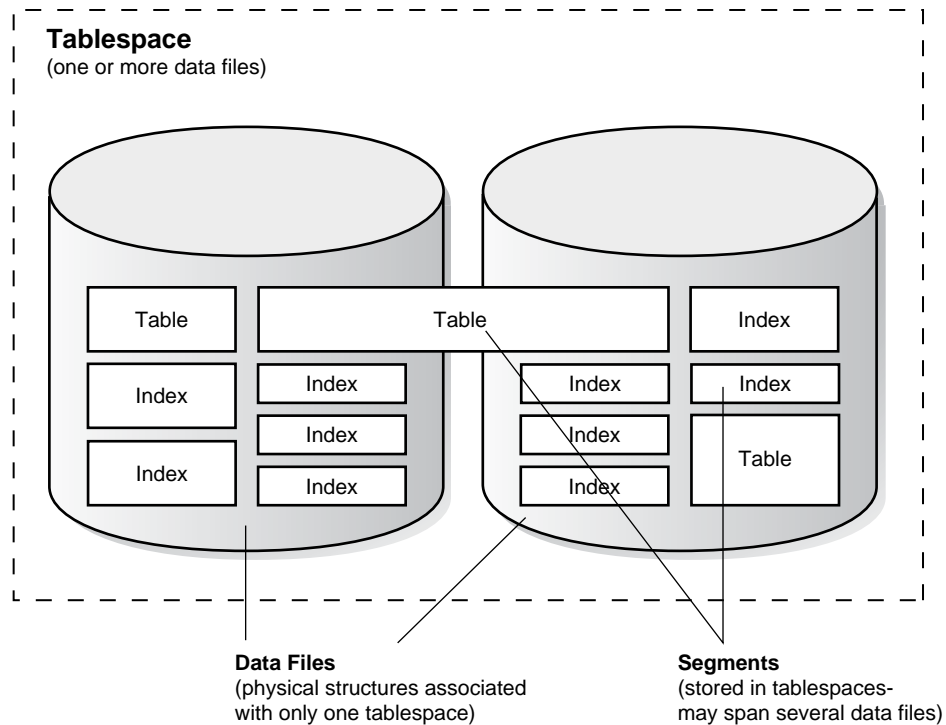
For ease of administration, Oracle Database allocates space for user data in **tablespaces**, which like segments are logical storage structures. Each segment belongs to only one tablespace. For example, the data for a nonpartitioned table is stored in a single segment, which in turn is stored in one tablespace.

Oracle Database physically stores tablespace data in data files. 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 database must have the `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`).

Figure 11–4 shows the relationship between tablespaces, data files, and segments.

Figure 11–4 Data Files and Tablespaces**See Also:**

- ["Overview of Tablespaces"](#) on page 12-30
- *Oracle Database Administrator's Guide* and *Oracle Database 2 Day DBA* 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.

Caution: 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"](#) on page 12-34
- *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. Offline data files cannot be accessed until they are brought back online.

Administrators may take data files offline for many reasons, including performing offline backups, renaming a data file, 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

See Also:

- ["Online and Offline Tablespaces"](#) on page 12-35
- *Oracle Database Administrator's Guide* to learn how to alter data file availability

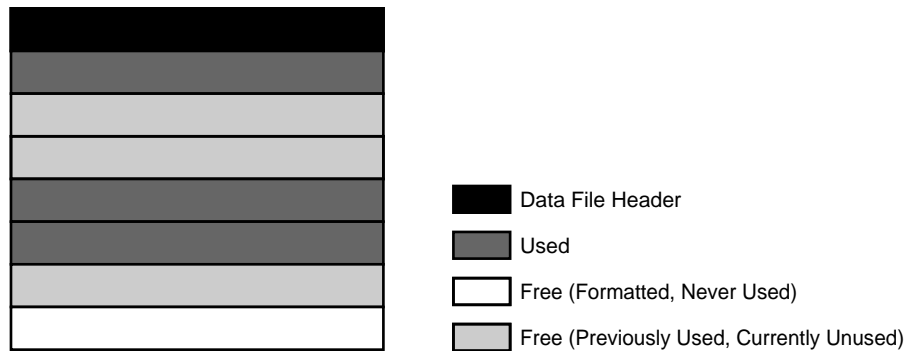
Data File Structure

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** and a **relative file number**. The absolute file number uniquely identifies the data file within the database. The relative file number 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.

[Figure 11-5](#) 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 referred to as **fragmented free space**.

Figure 11–5 Space in a Data File

See Also: *Oracle Database 2 Day DBA* and *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 database. Each database has one unique control file, although it may maintain identical copies of it.

Use of Control Files

The control file is the root file that Oracle Database uses to find 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 (see "[Overview of Instance Recovery](#)" on page 13-12). 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:

- ["Checkpoint Process \(CKPT\)"](#) on page 15-10
- *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 for 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, the database can be remounted and opened without [media recovery](#). If *all* control files of a database are lost, however, then the 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

These records contain 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

These records contain 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.

As explained in ["Overview of the Dynamic Performance Views"](#) on page 6-5, 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:

- *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 not yet written to the data files.

Oracle Database writes every transaction synchronously to the **redo log buffer**, which is then written to the online redo logs. The contents of the log include uncommitted transactions, undo data, and schema and object management statements.

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 (see "[Oracle LogMiner](#)" on page 18-8). Redo log files are a useful source of historical information about database activity.

See Also: "[Overview of Instance Recovery](#)" on page 13-12

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, two or more 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 is always available for writing while the other is being archived (if the database is in **ARCHIVELOG mode**).

See Also: *Oracle Database 2 Day + Real Application Clusters Guide* and *Oracle Real Application Clusters Administration and Deployment Guide* to learn about online redo log groups in Oracle RAC

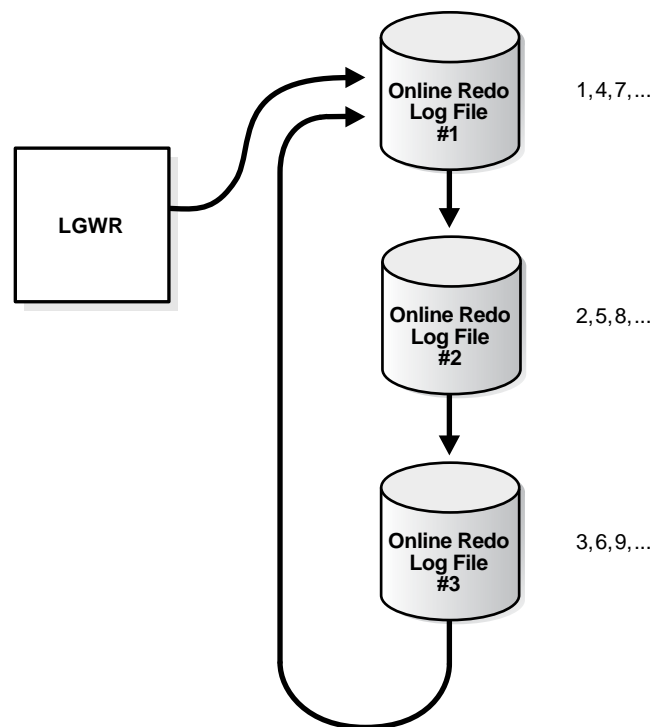
Online Redo Log Switches

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 (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 11–6](#) illustrates the circular writing of the redo log.

Figure 11–6 Reuse of Online Redo Log Files



The numbers in [Figure 11–6](#) 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 switch and log writer 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 (DBWn)**.
- 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. For example, an online redo log file may be **active** (required for instance

recovery) rather than **inactive** (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"](#) on page 15-7
- *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn how to manage the online redo log

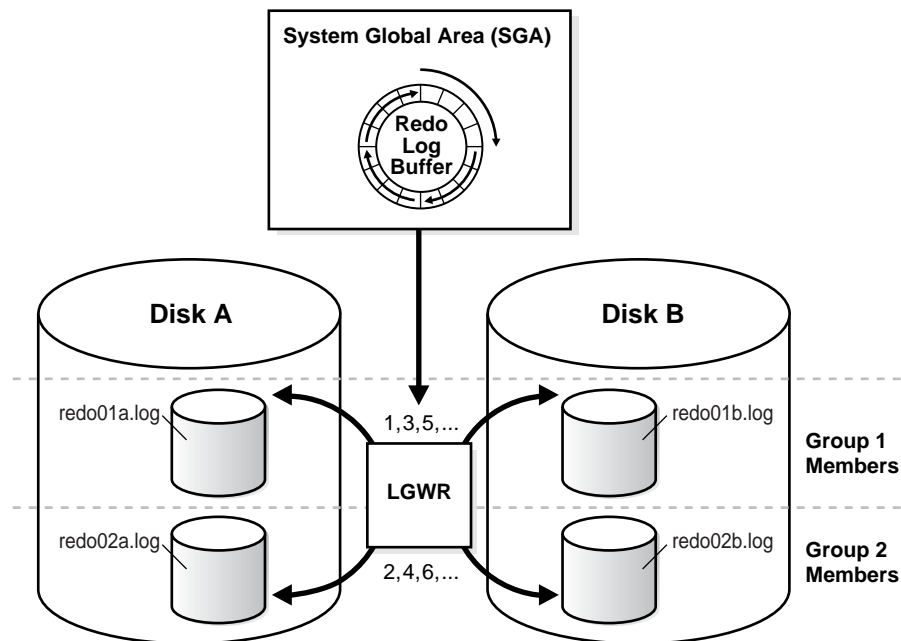
Multiple Copies of Online Redo Log Files

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 11-7](#), 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 11-7 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

Archived Redo Log Files

An **archived redo log file** is a copy of a filled member of an online redo log group. This file is not considered part of the database, but 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** (see "Computer Failures" on page 17-7)
- Obtain information about the history of a database using the LogMiner utility (see "Oracle LogMiner" on page 18-8)

Archiving is the operation of generating an archived redo log file. Archiving is either automatic or manual and 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 [Figure 11-7](#), 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 group created since you enabled archiving.

See Also:

- "Data File Recovery" on page 18-14
- *Oracle Database Administrator's Guide* to learn how to manage the archived redo log

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" on page 12-6

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:

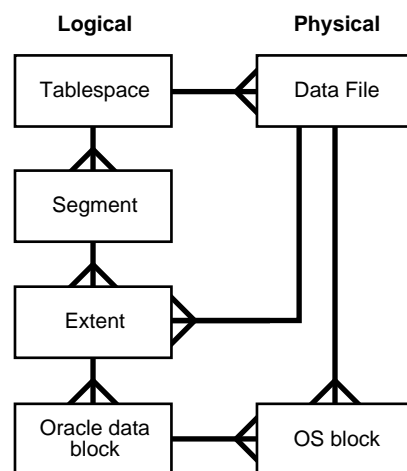
- [Introduction to Logical Storage Structures](#)
- [Overview of Data Blocks](#)
- [Overview of Extents](#)
- [Overview of Segments](#)
- [Overview of Tablespaces](#)

Introduction to Logical Storage Structures

Oracle Database allocates logical space for all data in the database. 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 (see [Chapter 11, "Physical Storage Structures"](#)). The data in the data files is stored in operating system blocks.

[Figure 12-1](#) is an entity-relationship diagram for physical and logical storage. The crow's foot notation represents a one-to-many relationship.

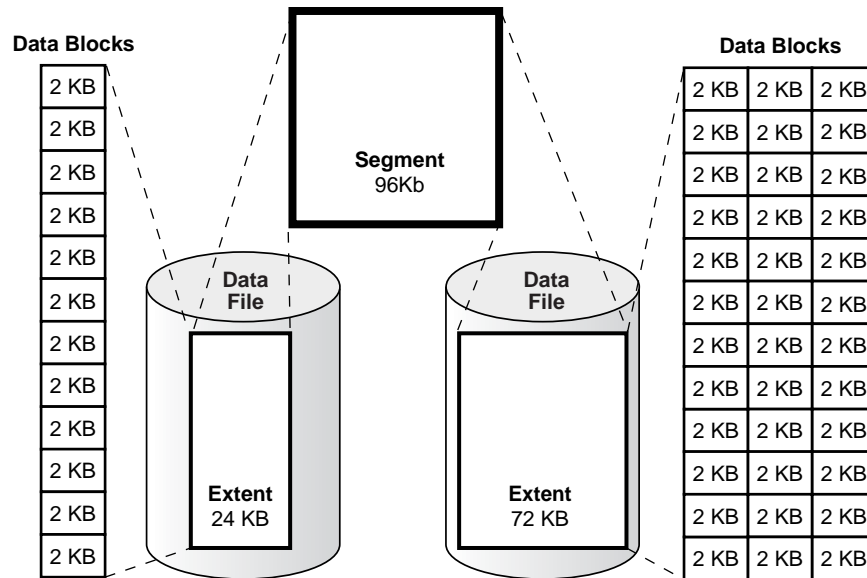
Figure 12-1 Logical and Physical Storage



Logical Storage Hierarchy

Figure 12–2 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 12–2 Segments, Extents, and Data Blocks Within a Tablespace



At the finest level of granularity, Oracle Database stores data in data blocks. One logical **data block** corresponds to a specific number of bytes of physical disk space, for example, 2 KB. Data blocks are the smallest units of storage that Oracle Database can use or allocate.

An **extent** is a set of logically contiguous data blocks allocated for storing a specific type of information. In Figure 12–2, 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.

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 Figure 12–2. 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: ["Overview of Data Files"](#) on page 11-7

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:

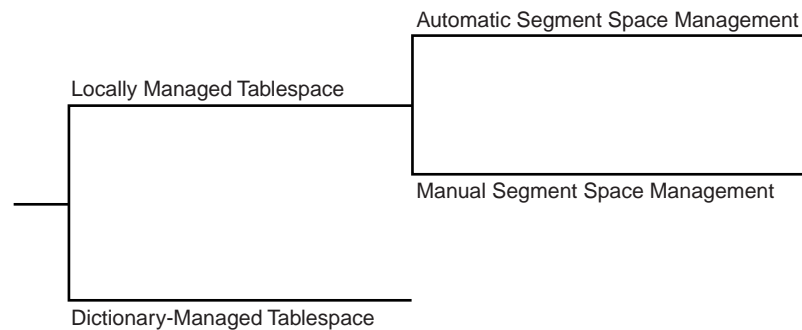
- Locally managed tablespaces (default)

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 (see "[Overview of the Data Dictionary](#)" on page 6-1).

Figure 12-3 shows the alternatives for logical space management in a tablespace.

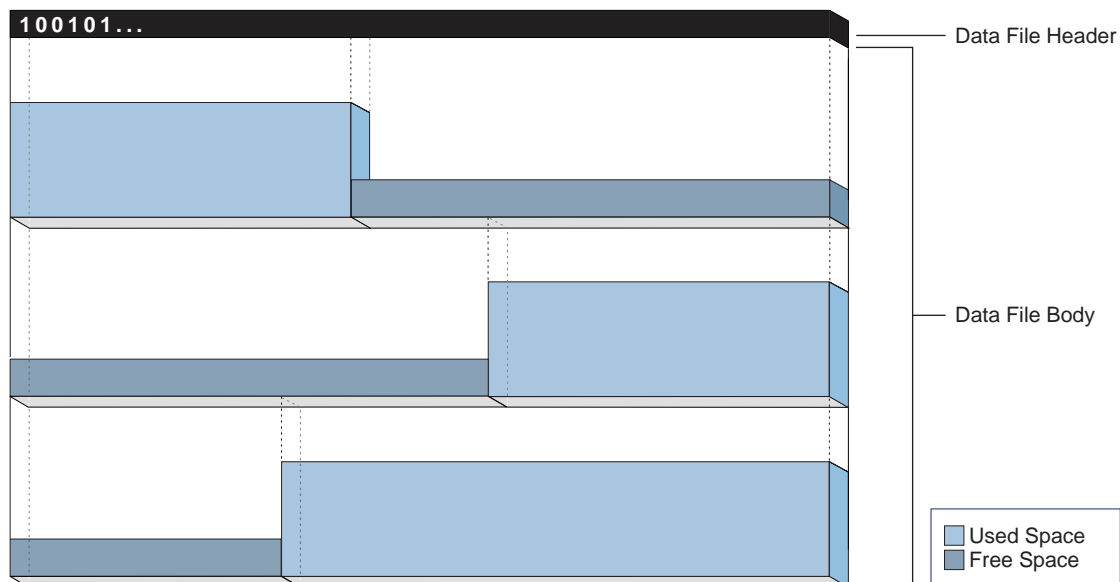
Figure 12-3 Logical Space Management



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. 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.



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 ASSM method uses bitmaps to manage space. 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](#)" on page 12-12).
- 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 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), 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`

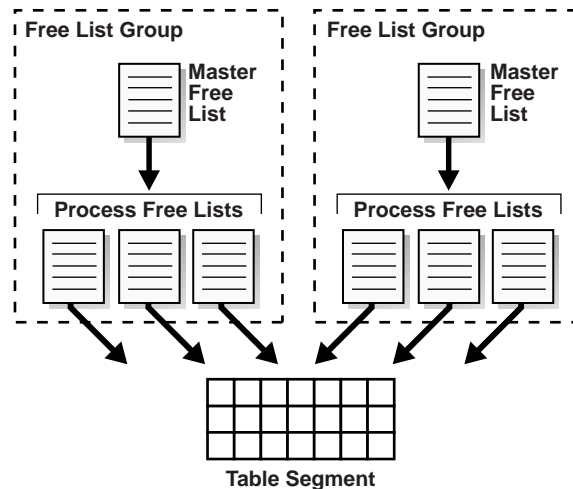
statement, then you cannot insert rows into a block in the segment until less than 40% of the block space is used.

As an illustration, 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 cannot fit in the block, and if the used space in the block is greater than or equal to `PCTUSED`, then the database takes the block off 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 12-4](#), 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 lists** in the group. Space overhead for free lists, especially for free list groups, can be significant.

Figure 12-4 Free List Groups



Managing segment space manually can be complex. You must adjust `PCTFREE` and `PCTUSED` to reduce row migration (see "[Chained and Migrated Rows](#)" on page 12-16) 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:

- *Oracle Database Administrator's Guide* to learn about locally managed tablespaces
- *Oracle Database 2 Day DBA and 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

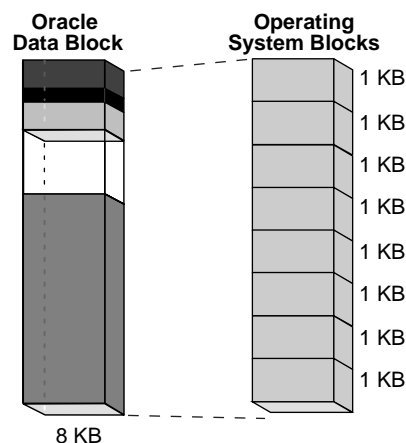
Oracle Database manages the logical storage space in the data files of a database in units called **data blocks**, also called **Oracle blocks** or **pages**. A data block is the minimum unit of database I/O.

Data Blocks and Operating System Blocks

At the physical level, database data is stored in disk files made up of operating system blocks. 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.

Figure 12–5 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 12–5 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**. 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

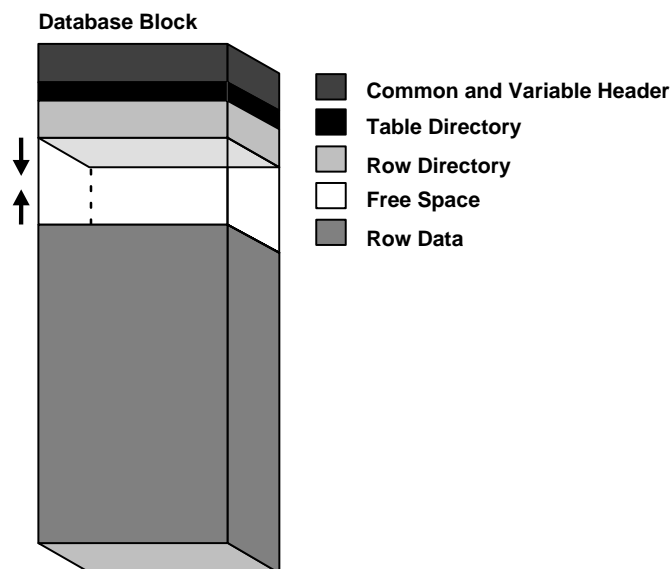
You can create individual tablespaces whose block size differs from the `DB_BLOCK_SIZE` setting. A nonstandard block size can be useful when moving a **transportable tablespace** to a different platform.

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. [Figure 12–6](#) shows the format of an uncompressed data block (see "[Data Block Compression](#)" on page 12-11 to learn about compressed blocks).

Figure 12–6 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. As shown in [Figure 12–6](#), 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. 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.

After space has been allocated in the row directory, the database does not reclaim this space after row deletion. 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 new rows are inserted in the block.

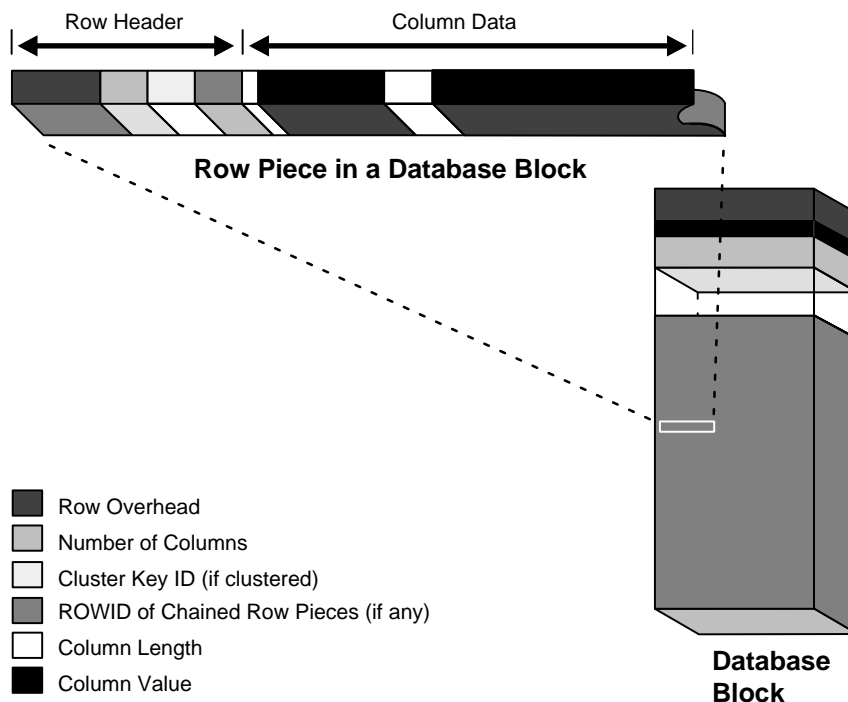
Some parts of the block overhead are fixed in size, but the total size is variable. On average, the block overhead totals 84 to 107 bytes.

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 **row pieces**. Each row piece has a **row header** and **column data**.

[Figure 12–7](#) shows the format of a row.

Figure 12-7 The Format of a Row Piece

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 (see ["Chained and Migrated Rows"](#) on page 12-16). A data block usually contains only one row piece per row.
- Cluster keys for **table clusters** (see ["Overview of Table Clusters"](#) on page 2-22)

A row fully contained in one block has at least 3 bytes of row header.

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 [Figure 12-7](#), 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"](#) on page 2-18 and ["Index Storage"](#) on page 3-20

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 12-1 queries the ROWID **pseudocolumn** to show the extended rowid of the row in the employees table for employee 100.

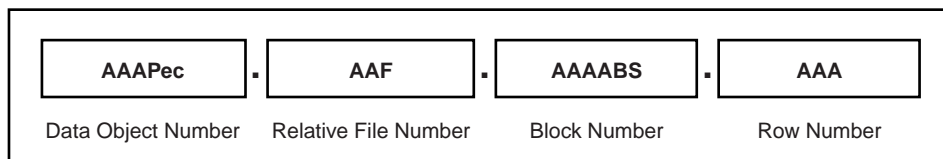
Example 12-1 ROWID Pseudocolumn

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

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

Figure 12-8 illustrates the format of an extended rowid.

Figure 12-8 ROWID Format



An extended rowid is displayed in a four-piece format, 000000FFFBBBBBBRRR, with the format divided into the following components:

- 000000
The **data object number** identifies the segment (data object AAAPec in Example 12-1). 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 Example 12-1).
- BBBBBB
The **data block number** identifies the block that contains the row (block AAAABS in Example 12-1). 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 Example 12-1).

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" on page 2-13
- *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 (see "Table Compression" on page 2-19). This section describes the format of data blocks that use compression.

The format of a data block that uses basic and OLTP table 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.

Assume that the rows in [Example 12-2](#) are stored in a data block for the seven-column sales table.

Example 12-2 Rows in 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 or OLTP table compression is applied to this table, the database replaces duplicate values with a symbol reference. [Example 12-3](#) is a conceptual representation of the compression in which the symbol * replaces 29-NOV-00 and % replaces 9999.

Example 12-3 OLTP Compressed Rows in sales Table

```
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 12-1](#) conceptually represents the symbol table that maps symbols to values.

Table 12-1 Symbol Table

Symbol	Value	Column	Rows
*	29-NOV-00	3	958-960
%	9999	5	956-960

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. This free space can also shrink

during updates, as when changing a trailing null to a nonnull 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

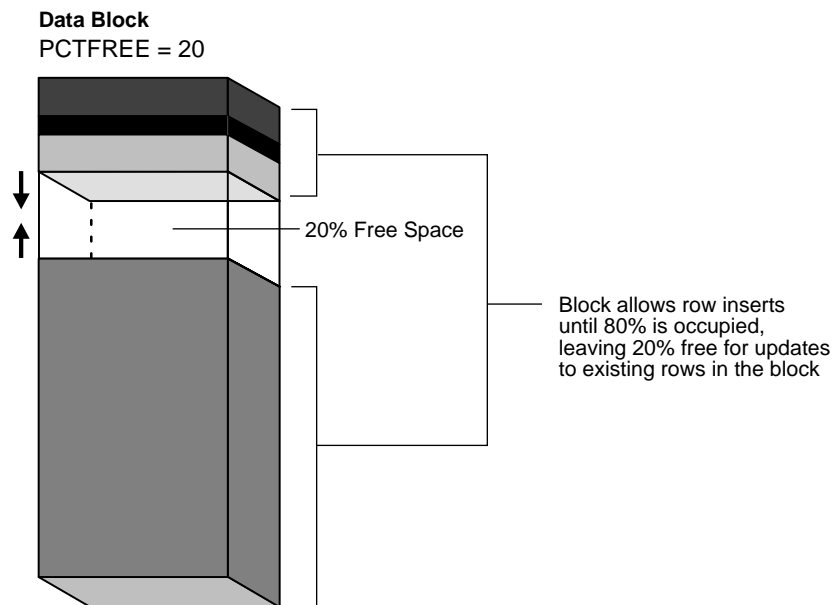
The PCTFREE storage parameter is essential to how the database manages free space. This SQL parameter sets the minimum percentage of a data block reserved as free space for updates to existing rows. Thus, PCTFREE is important for preventing row migration and avoiding wasted space.

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 12-9 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 12-9 PCTFREE



Note: This discussion does not apply to **LOB** data types, which do not use the PCTFREE storage parameter or free lists. See "[Overview of LOBs](#)" on page 19-12.

See Also: *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, a `PCTFREE` setting of 20% prevents the total amount of free space from dropping to 5% of the block, but permits 50% of the block to be free space. The following SQL statements can increase free 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 OLTP compression

If inserts fill a block with data, then the database invokes block compression, which may result in the block having more free space.

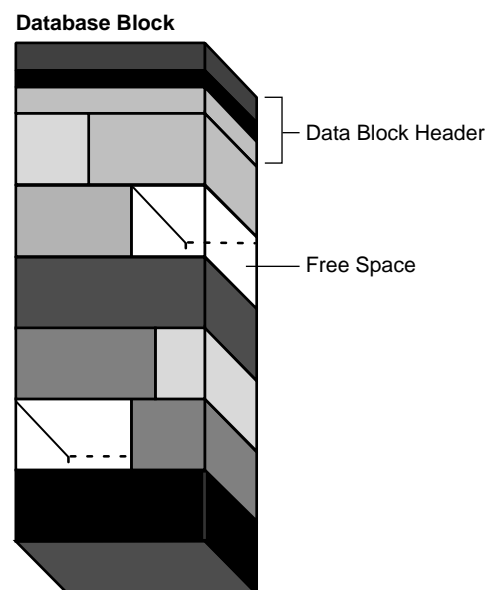
The space released is available for `INSERT` statements under the following conditions:

- If the `INSERT` statement is in the same transaction and 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), then the statement can use the space made available only after the other transaction commits and only if the space is needed.

See Also: *Oracle Database Administrator's Guide* to learn about OLTP compression

Coalescing Fragmented Space Released space may or may not be contiguous with the main area of free space in a data block, as shown in [Figure 12–10](#). Noncontiguous free space is called **fragmented space**.

Figure 12–10 Data Block with Fragmented Space



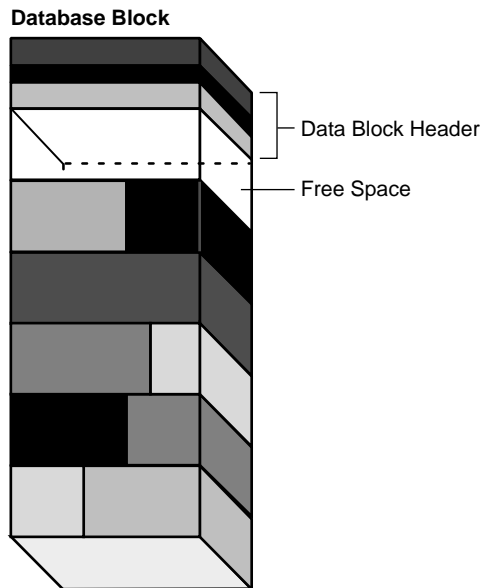
Oracle Database automatically and transparently coalesces the free space of a data block *only* when the following conditions are true:

- 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 12–11](#) shows a data block after space has been coalesced.

Figure 12–11 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.

Reuse of Index Space The database can reuse space within an index block. For example, if you insert a value into a column and delete it, and if an index exists on this column, then the database can reuse the index slot when a row requires it.

The database can reuse an index block itself. Unlike a table block, an index block only becomes free when it is empty. The database places the empty block on the free list of the index structure and makes it eligible for reuse. However, Oracle Database does not automatically compact the index: an `ALTER INDEX REBUILD` or `COALESCE` statement is required.

[Figure 12–12](#) represents 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 12-12 Index Before Coalescing

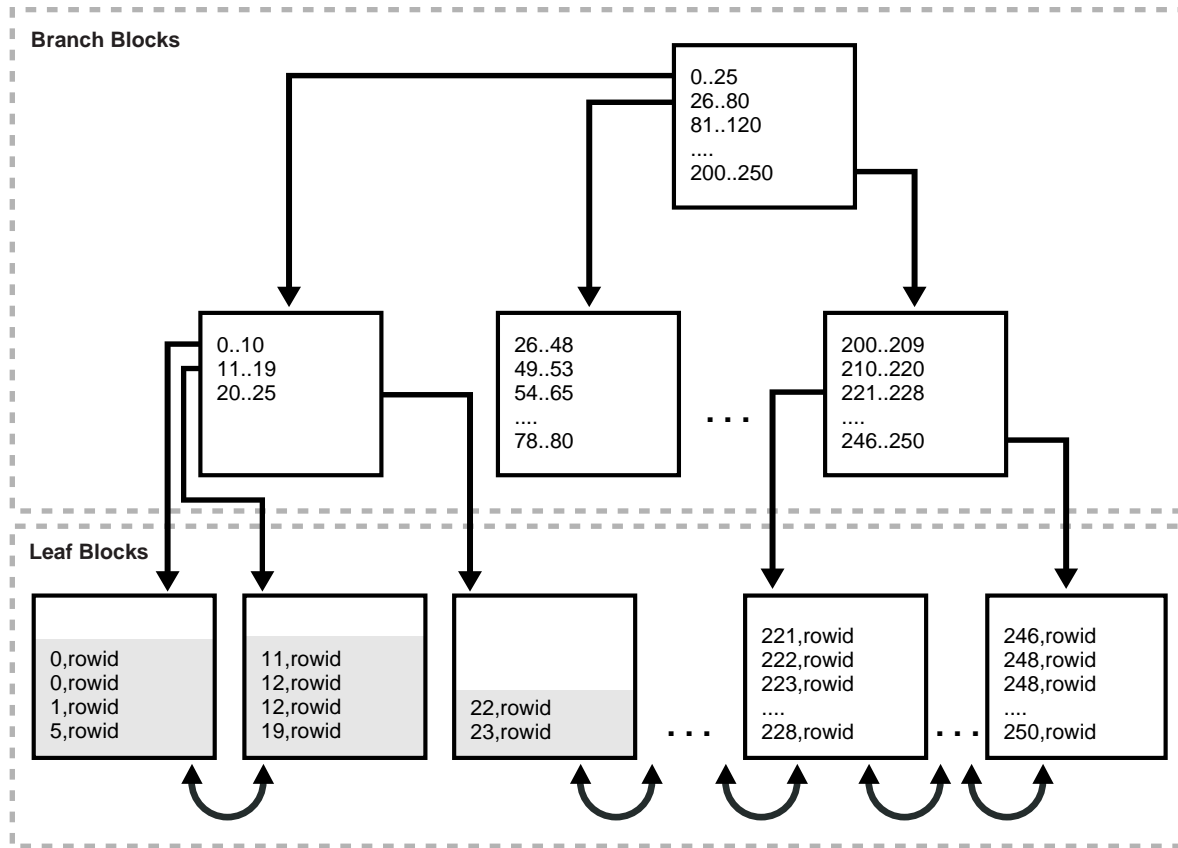
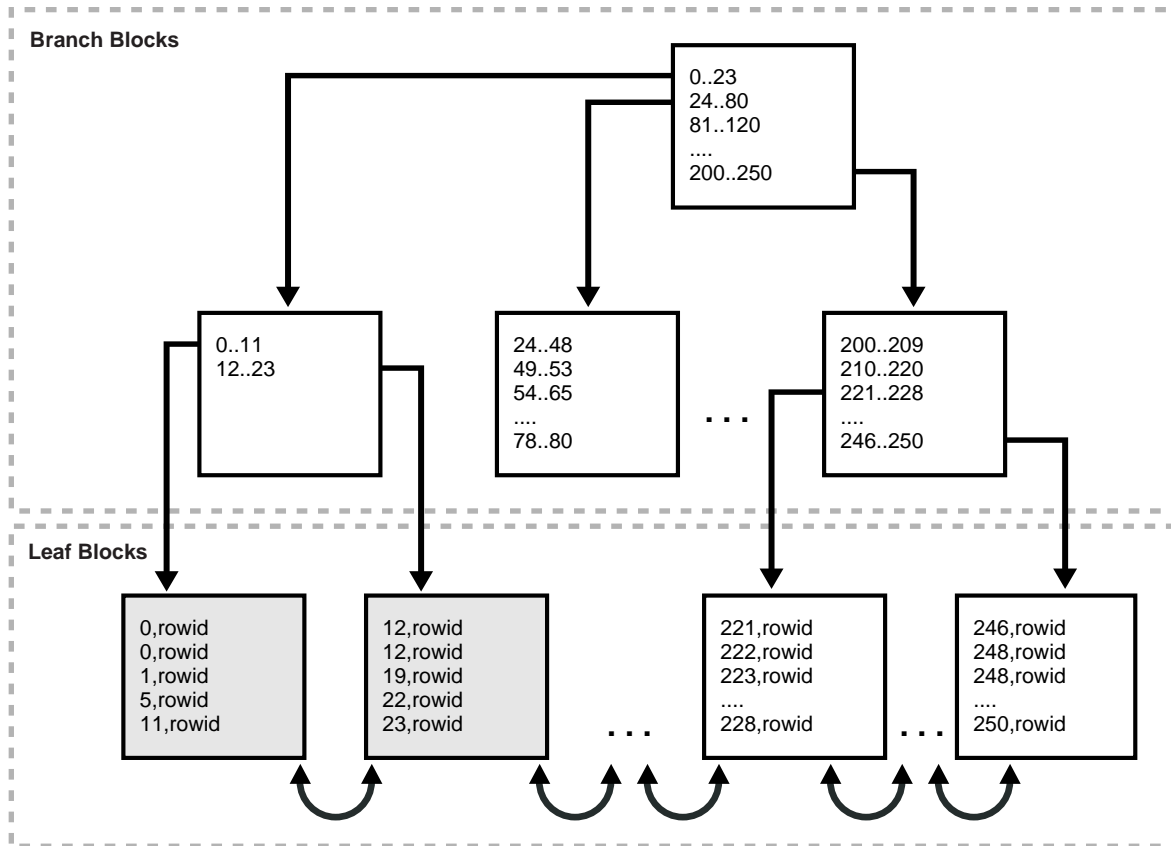


Figure 12-13 shows the index in Figure 12-12 after the index has been coalesced. The first two leaf blocks are now full, as indicated by the gray fill lines, and the third leaf block has been freed.

Figure 12–13 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

Chained and Migrated Rows

Oracle Database must manage rows that are too large to fit into a single block. 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, a VARCHAR2(4000) column in a 2 KB block, 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 12-14 depicts shows 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 12-14 Row Chaining

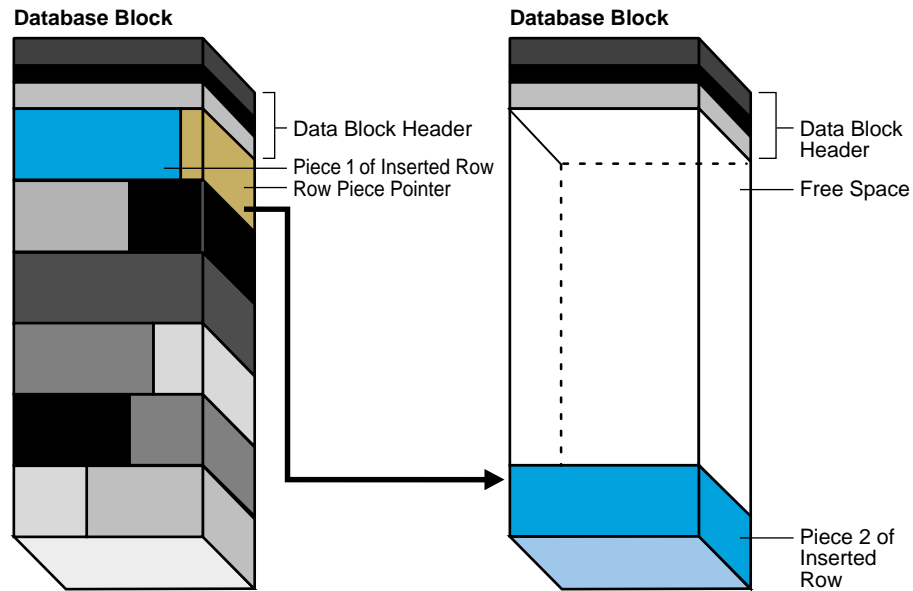
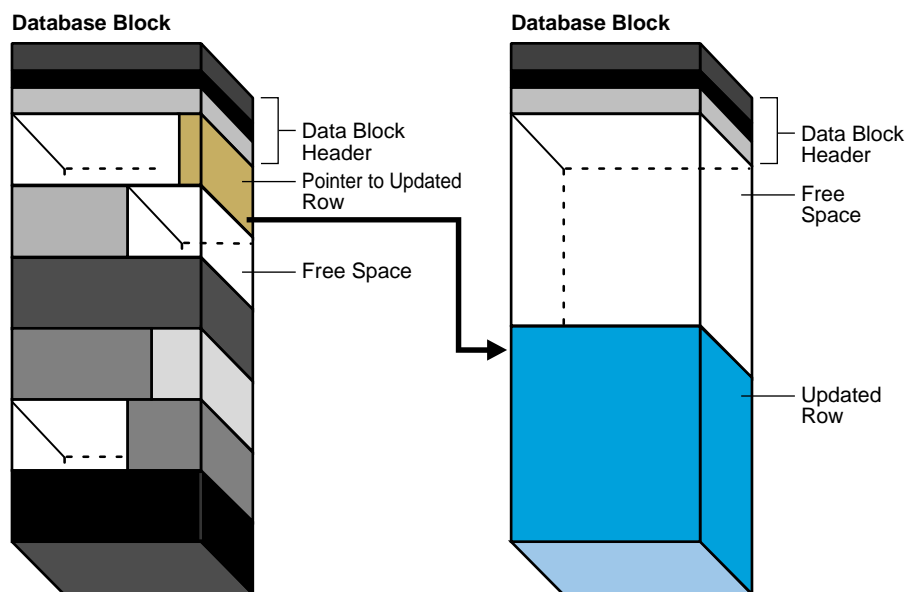


Figure 12-15, the left block contains a row that is updated so that the row is now too large for the block. The database moves the entire row to the right block and leaves a pointer to the migrated row in the left block.

Figure 12-15 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"](#) on page 2-19 and ["Rowids of Row Pieces"](#) on page 2-19
- *Oracle Database 2 Day DBA* and *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 Extents

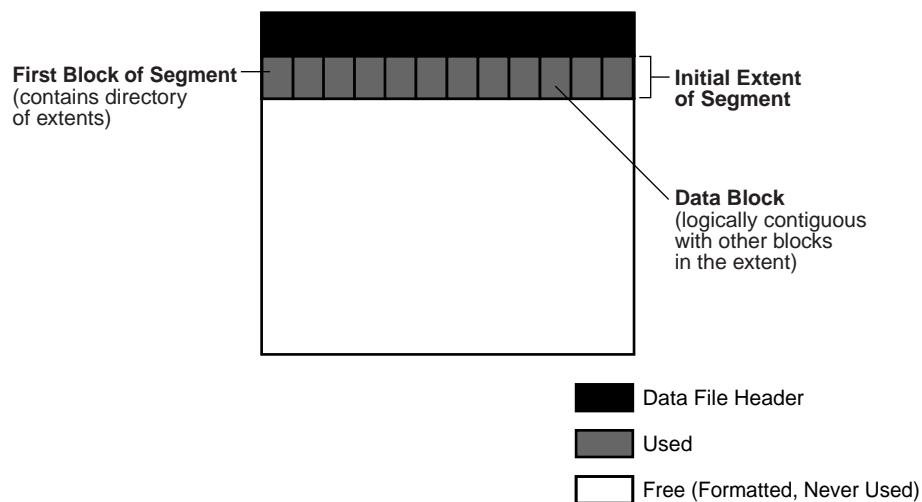
An **extent** is a logical unit of database storage space allocation made up of contiguous data blocks. Data blocks in an extent are logically contiguous but can be physically spread out on disk because of RAID striping and file system implementations.

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.

Although no data has been added to the segment, the 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 12-16](#) shows the initial extent in a segment in a data file that previously contained no data.

Figure 12-16 Initial Extent of a Segment

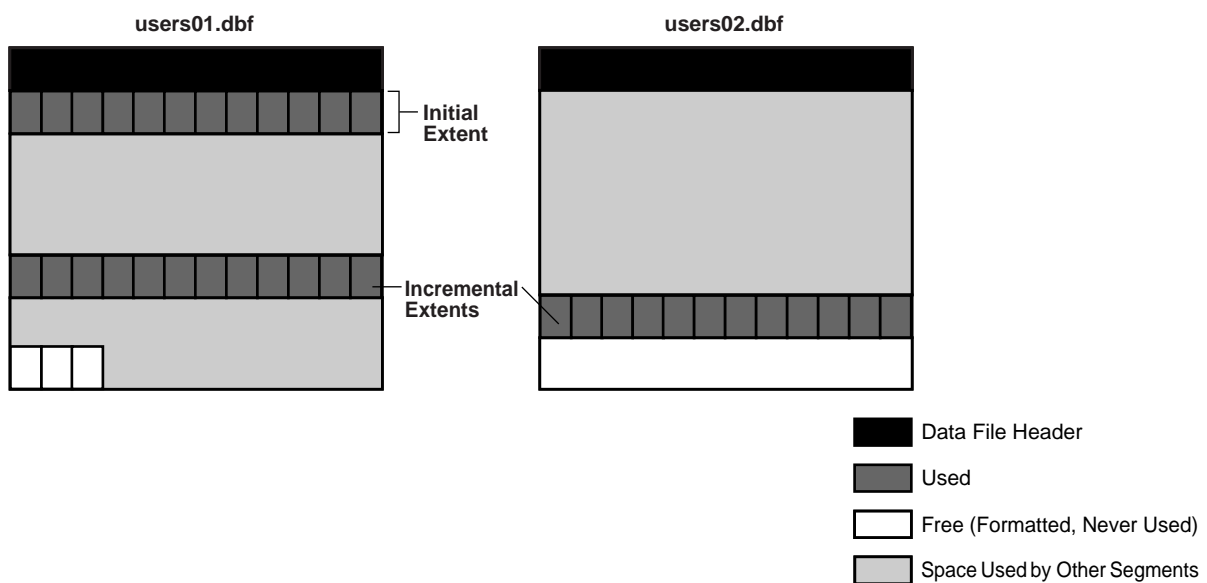


If the initial extent become full, and if more space is required, then the database automatically allocates an **incremental extent** for this segment. An incremental extent is a subsequent extent created for the segment.

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 12–17 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 allocate the next extent in `users01.dbf`.

Figure 12–17 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 (see "[Segment Space and the High Water Mark](#)" on page 12-27).

Note: This section applies to serial operations, in which one **server process** parses and runs a statement. Extents are allocated differently in parallel SQL statements, which entail multiple server processes.

See Also: *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` command. In Oracle Database 11g Release 2 (11.2.0.2), you can also drop the segment using the `DBMS_SPACE_ADMIN` package. 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.

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** (see "[Undo Tablespaces](#)" on page 12-33).

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:

- You can 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**.
- You can 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.
- You can rebuild or coalesce the index (see "[Reuse of Index Space](#)" on page 12-14).
- You can **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. 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.
- You can 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 (see "[Segment Space and the High Water Mark](#)" on page 12-27).

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: *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 setting 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 a table.

As explained in "[Logical Space Management](#)", 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. 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 and indexes. Starting in Oracle Database 11g Release 2 (11.2.0.2), the database also defers segment creation when creating 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 enables you to avoid 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.

You can use the DBMS_SPACE_ADMIN package to manage segments for empty objects. Starting with Oracle Database 11g Release 2 (11.2.0.2), you can use this PL/SQL package to do the following:

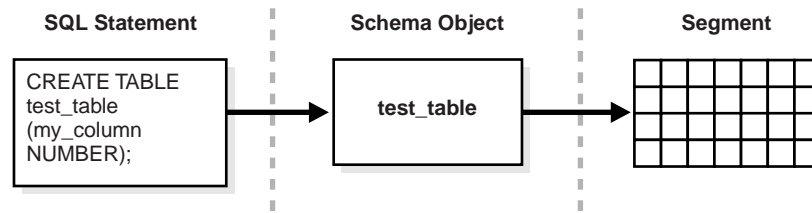
- Manually materialize segments for empty tables or partitions that do not have segments created
- Remove segments from empty tables or partitions that currently have an empty segment allocated

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 12–18](#), the database creates one segment for the table.

Figure 12–18 Creation of a User Segment

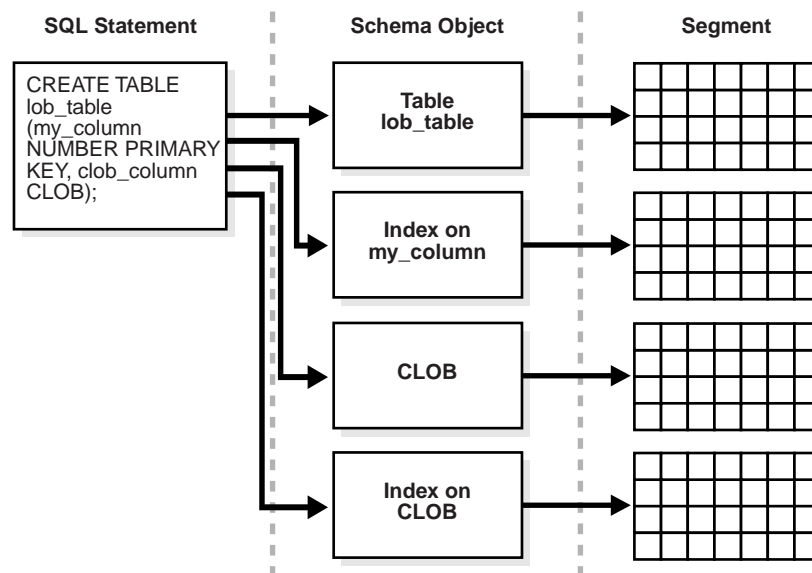


When you create a table with a [primary key](#) or unique key, Oracle Database automatically creates an index for this key. Again assume that deferred segment creation is disabled. You create a table as follows:

```
CREATE TABLE lob_table (my_column NUMBER PRIMARY KEY, clob_column CLOB);
```

[Figure 12–19](#) 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 (see ["Internal LOBs"](#) on page 19-12). Thus, the `CREATE TABLE` statement results in the creation of *four* different segments.

Figure 12–19 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 (see ["Storage Parameters for Extents"](#) on page 12-20). The parameters affect the efficiency of data retrieval and storage for the data segment associated with the object.

See Also:

- *Oracle Database Administrator's Guide* to learn how to manage deferred segment creation
- *Oracle Database Advanced Replication* for information on materialized views and materialized view logs
- *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 (see "[Overview of the Online Redo Log](#)" on page 11-12).

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:

- *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 Temporary Segments for Temporary Tables and Indexes

Oracle Database can also 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 the session and cannot access extents allocated for other sessions.

Oracle Database allocates segments for a temporary table when the first INSERT into that table occurs. The insertion can occur explicitly or because of CREATE TABLE AS SELECT. The first INSERT into a temporary table allocates the segments for the table and its indexes, creates the root page for the indexes, and allocates any LOB segments.

Segments for a temporary table are allocated in a temporary tablespace of the current user. Assume that 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:

- ["Temporary Tables"](#) on page 2-15
- *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 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. In this way, Oracle Database can efficiently access undo data without needing to read external logs.

Undo data 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.

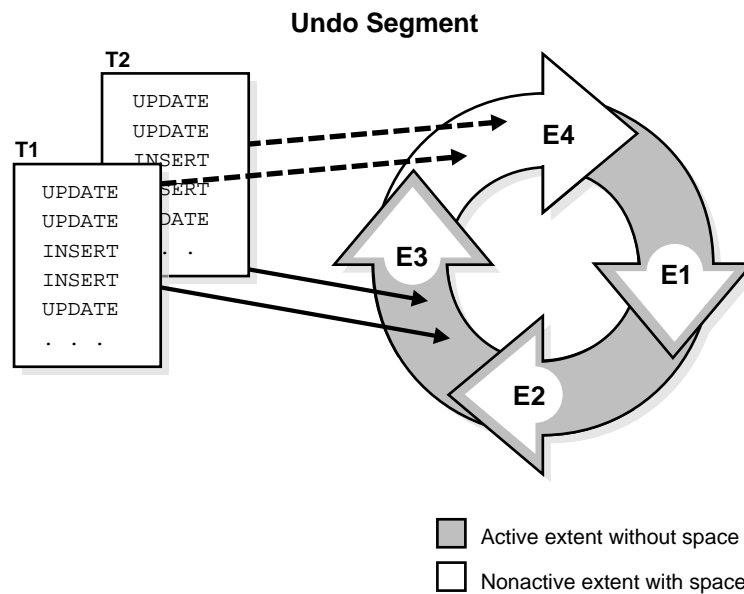
Undo Segments and Transactions

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 12–20](#) 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).

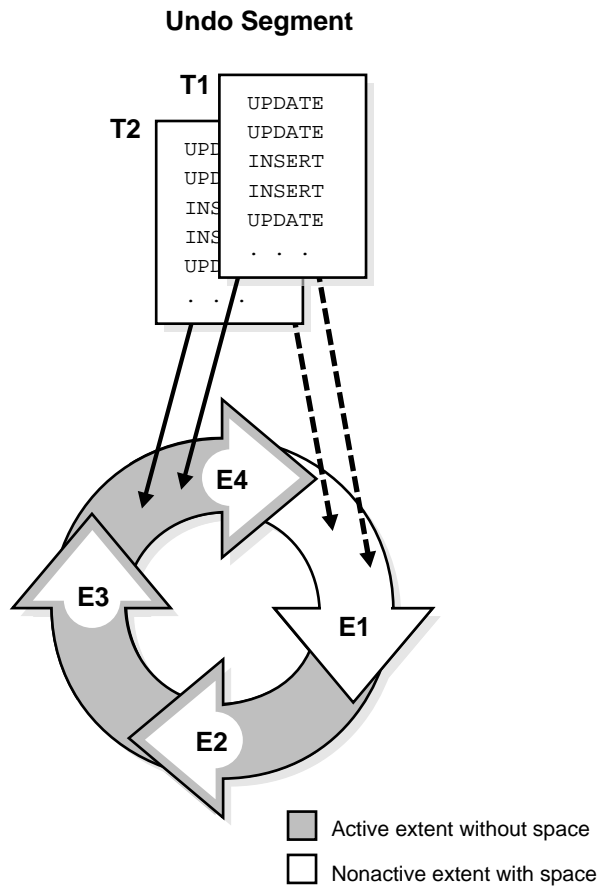
Figure 12-20 Ring of Allocated Extents in an Undo Segment



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 12-20](#) shows transactions T1 and T2 writing simultaneously to extent E3. Within an undo extent, a data block contains data for only one transaction.

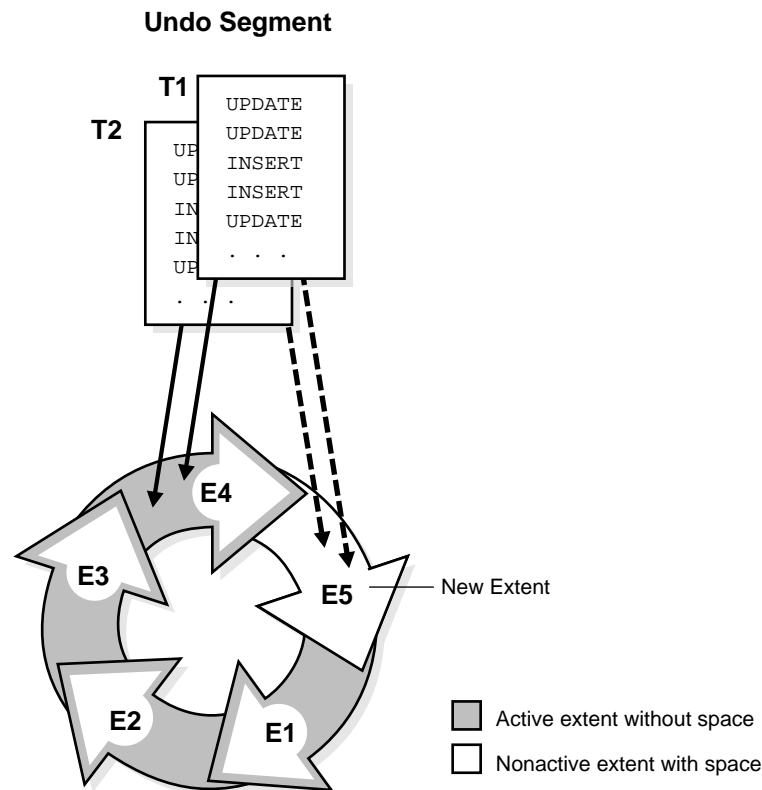
As the current undo extent fills, the first transaction needing space checks the availability of the next allocated extent in the ring. If the next extent does *not* contain data from an active transaction, then this extent becomes the current extent. Now all transactions that need space can write to the new current extent. In [Figure 12-21](#), when E4 is full, T1 and T2 continue writing to E1, overwriting the nonactive undo data in E1.

Figure 12–21 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 12–22](#) 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 12-22 Allocation of a New Extent for an Undo Segment



See Also: *Oracle Database 2 Day DBA* and *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.

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.

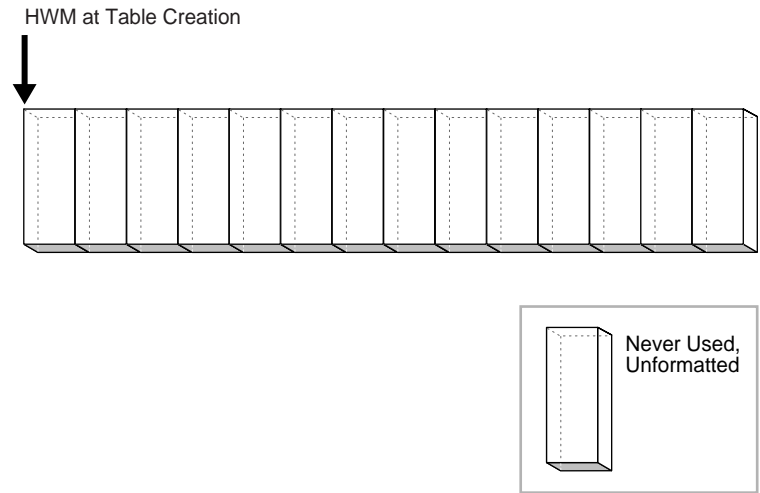
ASSM does not use free lists and so must manage space differently. When a session first inserts data into a table, the database formats a single bitmap block instead of preformatting a group of blocks as in MSSM. The bitmap tracks the state of blocks in the segment, taking the place of the free list. 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

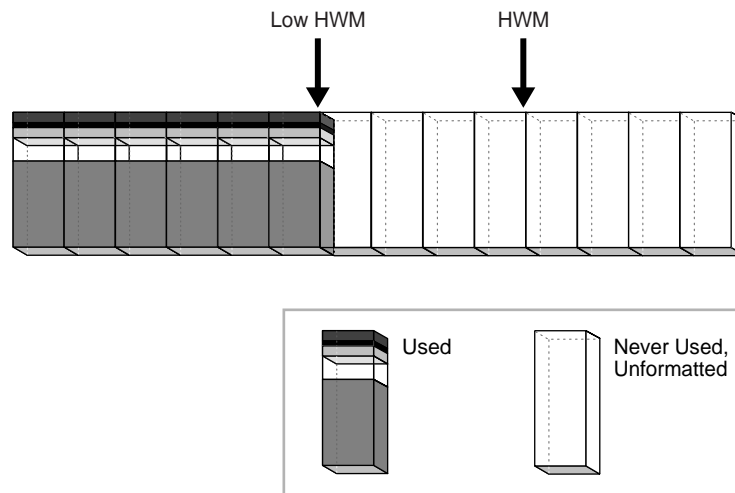
Figure 12–23 depicts an ASSM segment as a horizontal series of blocks. At table creation, the HWM is at the beginning of the segment on the left. Because no data has been inserted yet, all blocks in the segment are unformatted and never used.

Figure 12–23 HWM at Table Creation

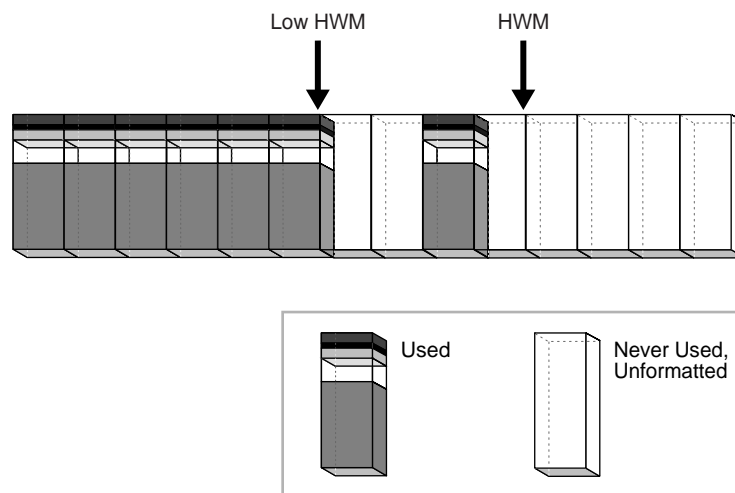


Suppose that a transaction inserts rows into the segment. The database must allocate a group of blocks to hold the rows. The allocated blocks fall below the HWM. The database formats a bitmap block in this group to hold the metadata, but does not preformat the remaining blocks in the group.

In Figure 12–24, 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.

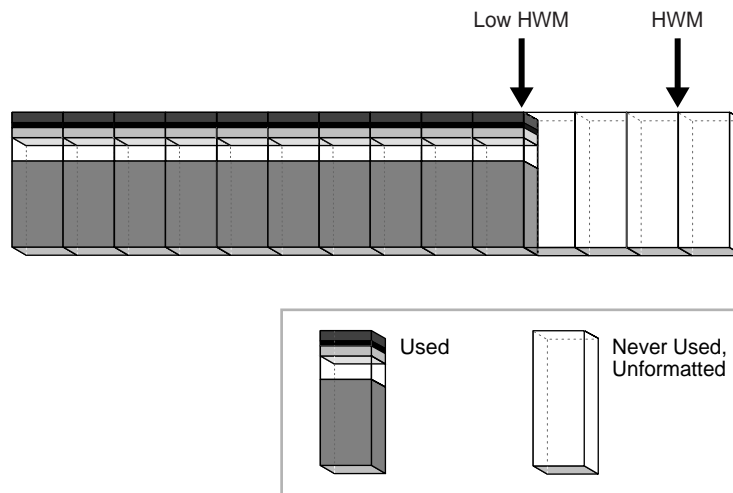
Figure 12–24 HWM and Low HWM

In [Figure 12–25](#), 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 12–25](#), the blocks to either side of the newly filled block are unformatted.

Figure 12–25 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 12–25](#). 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 12–26](#), the database advances the HWM to the right, allocating a new group of unformatted blocks.

Figure 12–26 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:

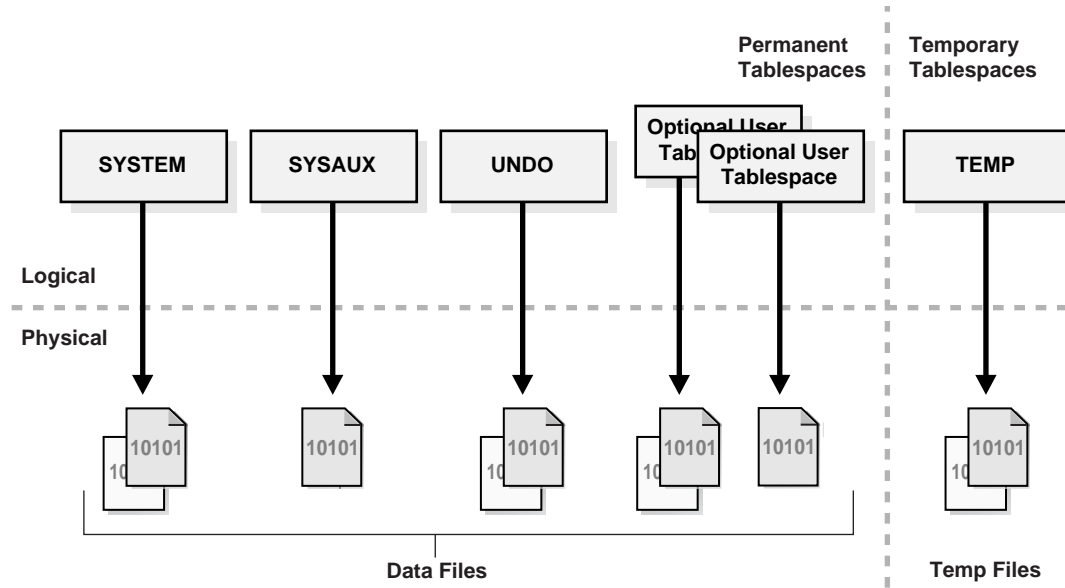
- *Oracle Database Administrator's Guide* to learn how to shrink segments online
- *Oracle Database SQL Language Reference* for `TRUNCATE TABLE` syntax and semantics

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.

A database must have the `SYSTEM` and `SYSAUX` tablespaces. [Figure 12–27](#) shows the tablespaces in a typical database. The following sections describe the tablespace types.

Figure 12-27 Tablespaces



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 (see "[Oracle Data Pump Export and Import](#)" on page 18-7)
- 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.

See Also:

- *Oracle Database Administrator's Guide* to learn how to transport tablespaces
- *Oracle Database Utilities* to learn about Oracle Data Pump
- *Oracle Streams Concepts and Administration* for more information on ways to copy or transport files

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"](#) on page 12-35 for information about the permanent online condition of the SYSTEM tablespace
- ["Tools for Database Installation and Configuration"](#) on page 18-4 to learn about DBCA
- *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. The SYSAUX tablespace provides a centralized location for database metadata that does not reside in the SYSTEM tablespace. It reduces the number of tablespaces created by default, both in the seed database and in user-defined databases.

Several database components, including Oracle Enterprise Manager and Oracle Streams, use the SYSAUX tablespace as their default storage location. Therefore, the SYSAUX tablespace is created automatically during database creation or upgrade.

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 (see "[Undo Segments](#)" on page 12-24). 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 management mode**. This 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.

An undo tablespace is automatically created with a new installation of Oracle Database. Earlier versions of Oracle Database may not include an undo tablespace and use legacy rollback segments instead, known as **manual undo management mode**. When upgrading to Oracle Database 11g, you can enable automatic undo management mode and create an undo tablespace. 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
- *Oracle Database 2 Day DBA* for information on the Undo Advisor and on how to use advisors

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

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. The database stores temporary tablespace data in **temp files**.

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.

When the `SYSTEM` tablespace is locally managed, a default temporary tablespace is included in the database by default during database creation. A locally managed `SYSTEM` tablespace cannot serve as default temporary storage.

Note: You cannot make a default temporary tablespace permanent.

You can specify a user-named default temporary tablespace when you create a database by using the `DEFAULT TEMPORARY TABLESPACE` extension to the `CREATE DATABASE` statement. If `SYSTEM` is dictionary managed, and if a default temporary tablespace is not defined at database creation, then `SYSTEM` is the default temporary storage. However, the database writes a warning in the **alert log** saying that a default temporary tablespace is recommended.

See Also:

- ["Permanent and Temporary Data Files"](#) on page 11-8
- *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`

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. 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 (DBWn)** 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](#)" on page 11-9
- "[Database Writer Process \(DBWn\)](#)" on page 15-8
- *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 tablespaces can do the following:

- Increase the storage capacity of a database

The maximum number of data files in a database is limited (usually to 64 KB files), 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:

- ["Backup and Recovery"](#) on page 18-9
- *Oracle Database Administrator's Guide* to learn how to manage bigfile tablespaces

Part V

Oracle Instance Architecture

This part describes the basic structural architecture of the Oracle database instance. This part contains the following chapters:

- [Chapter 13, "Oracle Database Instance"](#)
- [Chapter 14, "Memory Architecture"](#)
- [Chapter 15, "Process Architecture"](#)
- [Chapter 16, "Application and Networking 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.

This chapter contains the following sections:

- [Introduction to the Oracle Database Instance](#)
- [Overview of 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. A **database** is a set of physical files on disk created by the `CREATE DATABASE` statement. The instance manages its associated data and serves the users of the database.

Every running Oracle database is associated with at least one Oracle database instance. Because an instance exists in memory and a database exists 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**. The SGA serves various purposes, including the following:

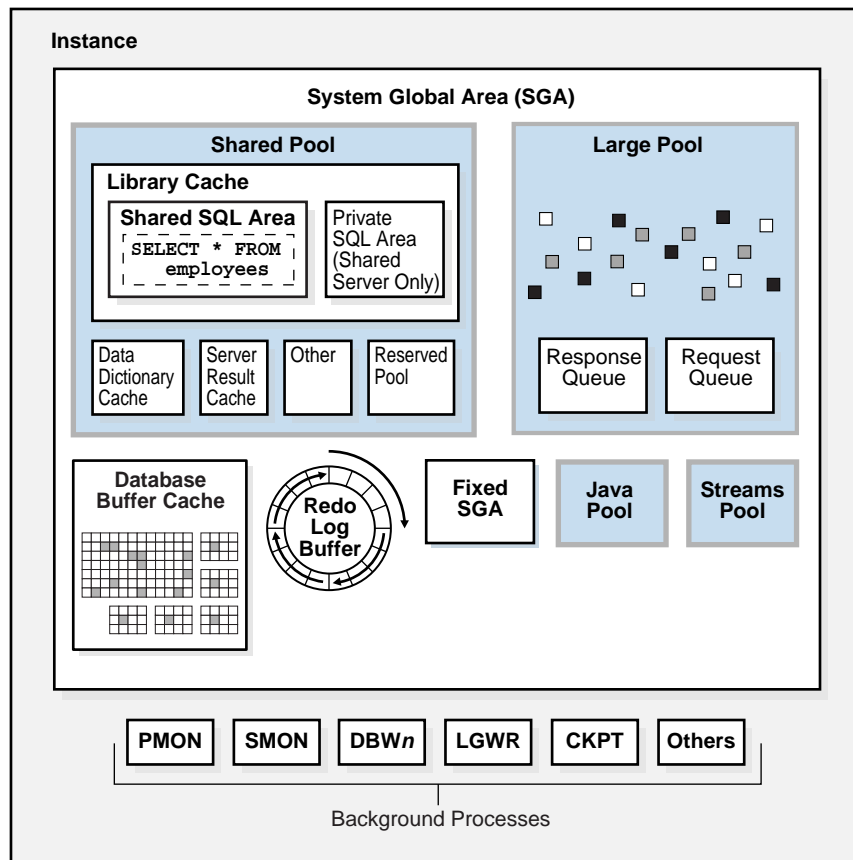
- Maintaining internal data structures that are accessed by many processes and threads concurrently
- Caching data blocks read from disk
- Buffering redo data before writing it to the online redo log files
- Storing SQL **execution plans**

The SGA is shared by the **Oracle processes**, which include **server processes** and background processes, running on a single computer. The way in which Oracle processes are associated with the SGA varies according to operating system.

A database instance includes background processes. Server processes, and the process memory allocated in these processes, also exist in the instance. The instance continues to function when server processes terminate.

Figure 13–1 shows the main components of an Oracle database instance.

Figure 13–1 Database Instance



See Also:

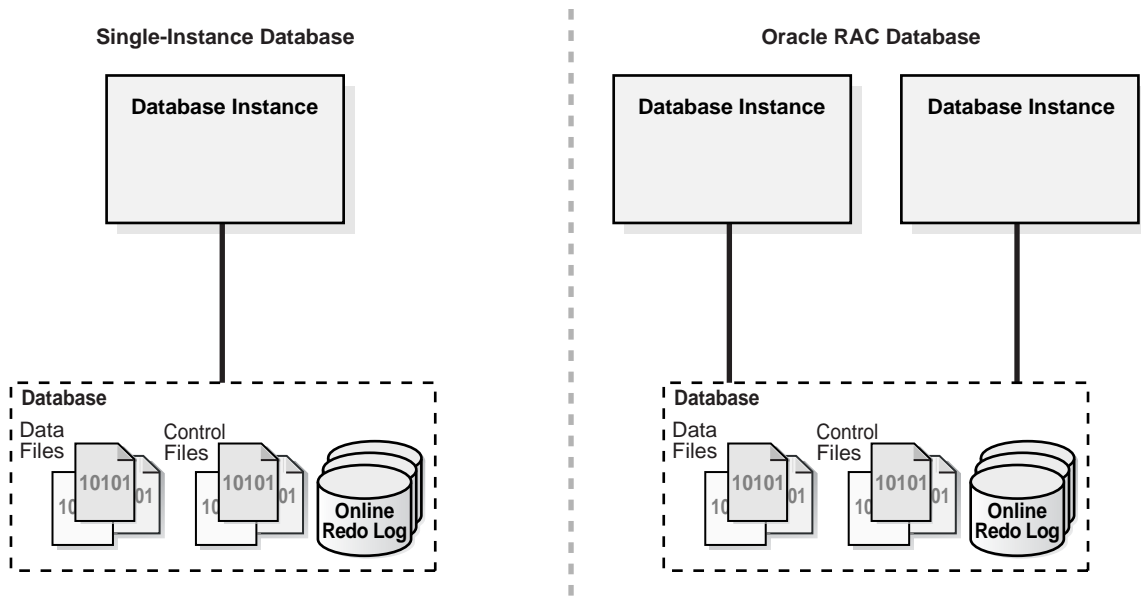
- ["Overview of the System Global Area"](#) on page 14-8
- ["Overview of Background Processes"](#) on page 15-7

Database Instance Configurations

You can run Oracle Database in either of the following mutually exclusive configurations:

- **Single-instance configuration**
A one-to-one relationship exists between the database and an instance.
- **Oracle Real Application Clusters (Oracle RAC) configuration**
A one-to-many relationship exists between the database and instances.

Figure 13–2 shows possible database instance configurations.

Figure 13–2 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

Duration of an Instance

An instance begins when it is created with the `STARTUP` command and ends when it is terminated. During this period, an 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.

[Table 13–1](#) illustrates a database instance attempting to reopen a database that it previously closed.

Table 13–1 Duration of an Instance

Statement	Explanation
<pre>SQL> STARTUP ORACLE instance started. Total System Global Area 468729856 bytes Fixed Size 1333556 bytes Variable Size 440403660 bytes Database Buffers 16777216 bytes Redo Buffers 10215424 bytes Database mounted. Database opened.</pre>	The <code>STARTUP</code> command creates an instance, which mounts and opens the database.
<pre>SQL> SELECT TO_CHAR(STARTUP_TIME, 'MON-DD-RR HH24:MI:SS') AS "Inst Start Time" FROM V\$INSTANCE; Inst Start Time ----- JUN-18-09 13:14:48</pre>	This query shows the time that the current instance was started.
<pre>SQL> ALTER DATABASE CLOSE; Database altered.</pre>	The instance closes the database, leaving it in a mounted state. The instance can read and write to the control file but not the data files .
<pre>SQL> ALTER DATABASE OPEN; ALTER DATABASE OPEN * ERROR at line 1: ORA-16196: database has been previously opened and closed</pre>	The instance attempts to reopen the database that it previously closed. Oracle Database issues an error because the same instance cannot open a database twice.
<pre>SQL> SHUTDOWN IMMEDIATE</pre>	At this stage, the only option for the instance is to shut down, ending the life of this instance.
<pre>SQL> STARTUP Oracle instance started. . . .</pre>	The <code>STARTUP</code> command creates a new instance and mounts and open the database.
<pre>SQL> SELECT TO_CHAR(STARTUP_TIME, 'MON-DD-RR HH24:MI:SS') AS "Inst Start Time" FROM V\$INSTANCE; Inst Start Time ----- JUN-18-09 13:16:40</pre>	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.

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, the SID is used by default to locate the parameter file, which is used to locate relevant files such as the database control files.

On most platforms, the `ORACLE_SID` environment variable sets the SID, whereas the `ORACLE_HOME` variable sets the Oracle home. When connecting to an instance, clients can specify the SID in an Oracle Net connection or use a net service name. Oracle Database converts a service name into an `ORACLE_HOME` and `ORACLE_SID`.

See Also:

- ["Service Names"](#) on page 16-8
- *Oracle Database Administrator's Guide* to learn how to specify an Oracle SID

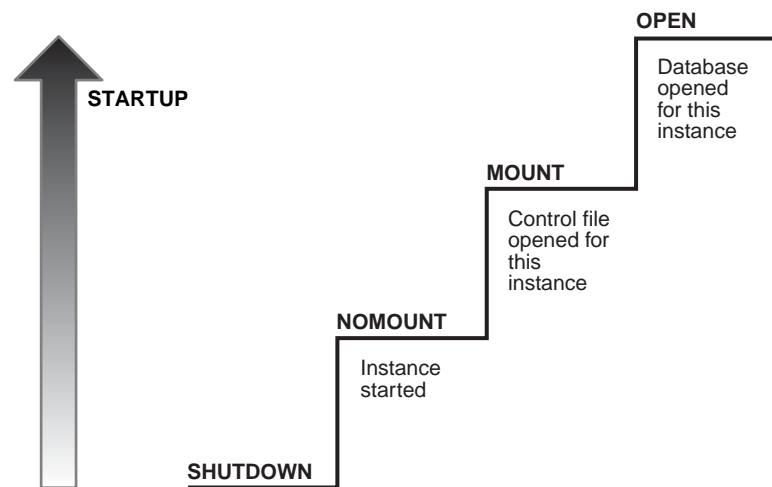
Overview of Instance Startup and Shutdown

A database instance provides user access to a database. This section explains the possible states of the instance and the database.

Overview of Instance and Database Startup

In a typical use case, 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. [Figure 13-3](#) shows how a database progresses from a shutdown state to an open state.

Figure 13-3 Instance and Database Startup Sequence



A database goes through the following phases when it proceeds from a shutdown state to an open database state:

1. Instance started without mounting database
The instance is started, but is not yet associated with a database.
["How an Instance Is Started"](#) on page 13-6 explains this stage.
2. Database mounted
The instance is started and is associated with a database by reading its **control file** (see ["Overview of Control Files"](#) on page 11-10). The database is closed to users.
["How a Database Is Mounted"](#) on page 13-7 explains this stage.
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"](#) on page 13-7 explains this stage.

See Also:

- ["Oracle Enterprise Manager"](#) on page 18-2
- *Oracle Database 2 Day DBA* and *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 the `SYSDBA` or `SYSOPER` system privileges and the database uses password files to authenticate database administrators over the network.

`SYSDBA` and `SYSOPER` are special **system privileges** that enable access to a database instance even when the database is not open. Control of these privileges is outside of the database itself.

When you connect 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 Schemas"](#) on page 2-5
- ["Overview of Database Security"](#) on page 17-1 to learn about password files and authentication for database administrators
- *Oracle Database Administrator's Guide* to learn about `SYSDBA` and `SYSOPER`

How an Instance Is Started

When Oracle Database starts an instance, it performs the following basic steps:

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 operations. 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 (see ["Online and Offline Tablespaces"](#) on page 12-35), 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

See Also: ["Data Repair"](#) on page 18-12

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. See *Oracle Data Guard Concepts and Administration*.

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

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.

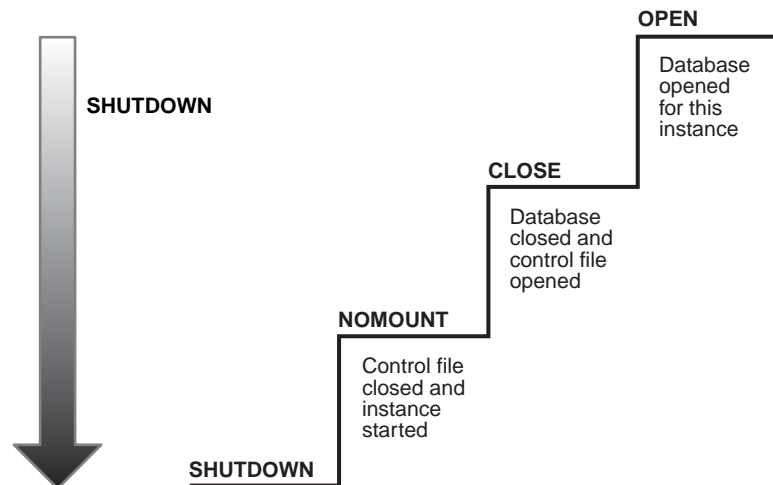
See Also: "[Backup and Recovery](#)" on page 18-9

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.

Figure 13-4 shows the progression from an open state to a consistent shutdown.

Figure 13-4 Instance and Database Shutdown Sequence



Oracle Database automatically performs the following steps whenever an open database is shut down consistently:

1. Database closed

The database is mounted, but online data files and redo log files are closed.

"[How a Database Is Closed](#)" explains this stage.

2. Database unmounted

The instance is started, but is no longer associated with the control file of the database.

"[How a Database Is Unmounted](#)" explains this stage.

3. Database instance shut down

The database instance is no longer started.

"[How an Instance Is Shut Down](#)" explains this stage.

Oracle Database does not go through all of the preceding steps in an instance failure or SHUTDOWN ABORT, which immediately terminates the instance.

See Also: *Oracle Database 2 Day DBA* and *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. [Table 13–2](#) summarizes the behavior of the different shutdown modes.

Table 13–2 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
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.

Note: 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.

- 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 2 Day DBA* and *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. Next, 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. 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 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 is shut down, the SGA is removed from memory and the background processes are terminated.

In unusual circumstances, shutdown of an 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 `SQL*Plus` or using Enterprise Manager.

See Also: *Oracle Database Administrator's Guide* for more detailed information about database shutdown

Overview of Checkpoints

A **checkpoint** is a crucial mechanism in consistent database shutdowns, instance recovery, and Oracle Database operation generally. The term **checkpoint** 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

See Also: "[System Change Numbers \(SCNs\)](#)" on page 10-5

Purpose of Checkpoints

Oracle Database uses checkpoints to achieve the following goals:

- Reduce the time required for recovery in case of an instance or media failure
- Ensure that dirty buffers in the buffer cache are written to disk regularly
- Ensure that all committed data is written 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. `DBWn` checks at least every three seconds to determine whether it has work to do. When `DBWn` 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 \(CKPT\)"](#) on page 15-10
- *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.

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 (LGWR)** writes both the remaining redo entries in memory and the transaction SCN to the **online redo log**. However, the **database writer (DBWn)** 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 database.
- 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 \(DBWn\)"](#) on page 15-8 and ["Database Buffer Cache"](#) on page 14-9
- ["Introduction to Data Concurrency and Consistency"](#) on page 9-1

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:

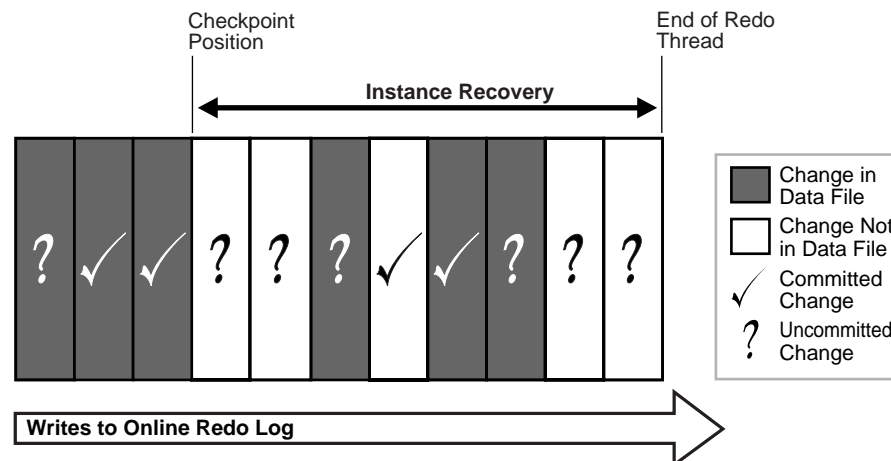
- ["System Monitor Process \(SMON\)"](#) on page 15-8
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn about instance recovery in an Oracle RAC database

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.

Figure 13–5 depicts the redo thread in the online redo log.

Figure 13–5 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 13–5, 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

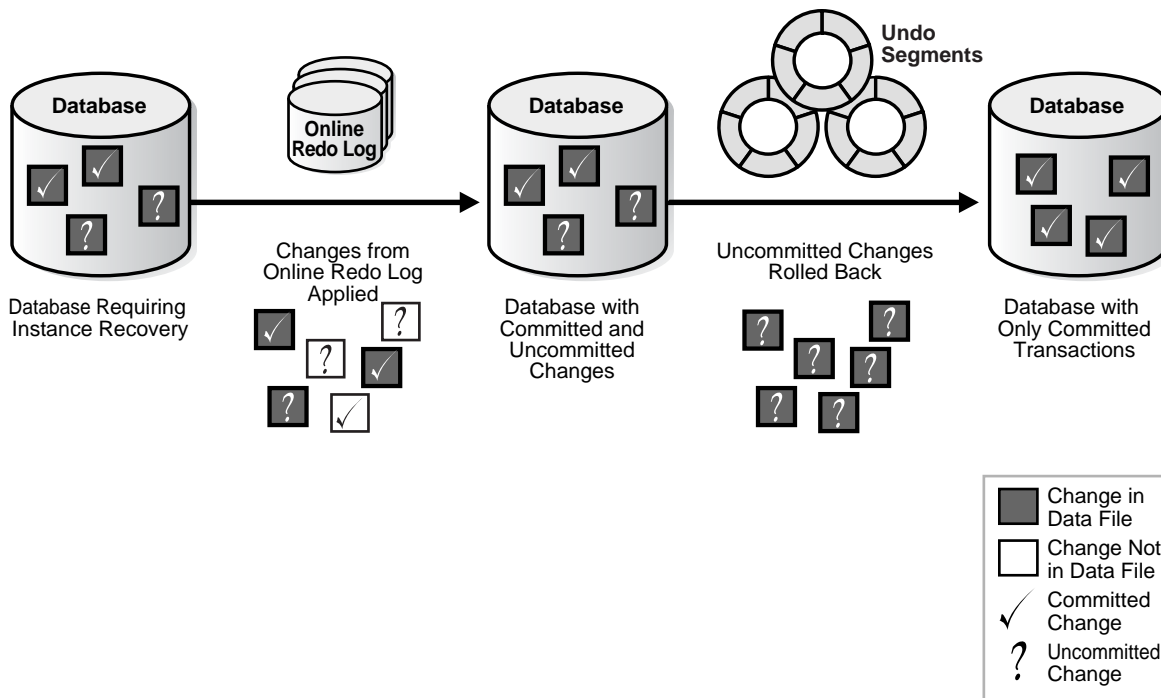
The first phase of instance recovery is called **cache recovery** or **rolling forward**, and involves reapplying all of the changes recorded in the online redo log to the data files. Because rollback data is recorded in the online redo log, rolling forward also regenerates the corresponding undo segments.

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**.

Figure 13–6 illustrates rolling forward and rolling back, the two steps necessary to recover from database instance failure.

Figure 13–6 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"](#) on page 12-24 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` command. 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.

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 adequate in most cases.

Functional Groups of Initialization Parameters

Most initialization parameters belong to one of the following functional 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. In most cases, 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 (see ["Tools for Database Installation and Configuration"](#) on page 18-4). 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 2 Day DBA* and *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 that is managed by Oracle Database. A server parameter file has the following key characteristics:

- Only one server parameter file exists for a database. This file must reside on the database host.
- The server parameter file is written to and read by only by Oracle Database, not by client applications.
- 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
```

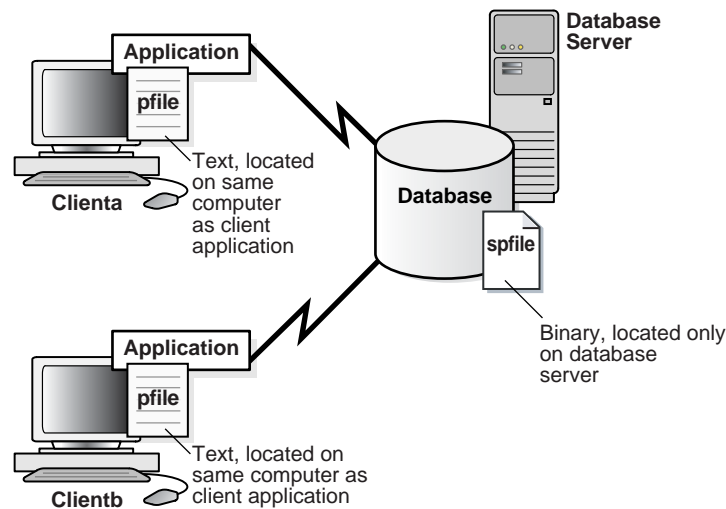
```

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 13-7](#). A server parameter file solves the problem of the proliferation of parameter files.

Figure 13-7 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. [Table 13-3](#) summarizes the differences.

Table 13-3 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

Table 13–3 (Cont.) Static and Dynamic Initialization Parameters

Characteristic	Static	Dynamic
Described as "Modifiable" in <i>Oracle Database Reference</i> 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 (see "[Locale-Specific Settings](#)" on page 19-10).

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 are written 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`
Changes are written 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 illegal values from being written to the server parameter file.

See Also:

- *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
- 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, 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. An instance can create ADR before a database has been created.

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.

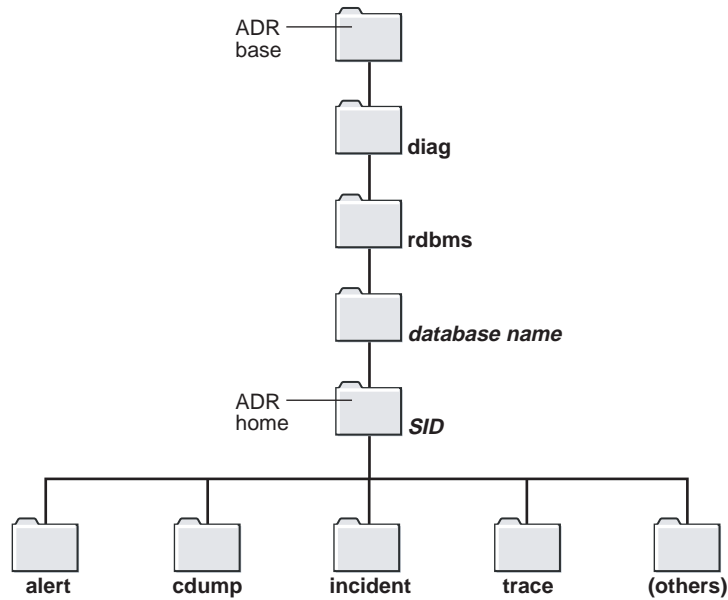
See Also: *Oracle Database Administrator's Guide* for detailed information about the fault diagnosability infrastructure

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 ASM, each database instance and each ASM instance has its own ADR home.

[Figure 13-8](#) illustrates the ADR directory hierarchy for a database instance. Other ADR homes for other Oracle products or components, such as ASM or Oracle Net Services, can exist within this hierarchy, under the same ADR base.

Figure 13–8 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 13–1 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> SELECT NAME, VALUE FROM V$DIAG_INFO;
  
```

NAME	VALUE
Diag Enabled	TRUE
ADR Base	/u01/oracle/log
ADR Home	/u01/oracle/log/diag/rdbms/dbn/osi
Diag Trace	/u01/oracle/log/diag/rdbms/dbn/osi/trace
Diag Alert	/u01/oracle/log/diag/rdbms/dbn/osi/alert
Diag Incident	/u01/oracle/log/diag/rdbms/dbn/osi/incident
Diag Cdump	/u01/oracle/log/diag/rdbms/dbn/osi/cdump
Health Monitor	/u01/oracle/log/diag/rdbms/dbn/osi/hm
Default Trace File	/u01/oracle/log/diag/rdbms/dbn/osi/trace/osi_ora_10533.trc

Active Problem Count 0
Active Incident Count 0

The following sections describe the contents of ADR.

Alert Log

Each database has an **alert log**, which is an XML 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 **DDL** statements and 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

Oracle Database uses the alert log as an alternative to displaying information in the Enterprise Manager GUI. 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 13-8](#) when you first start a database instance, even if no database has been created yet. The following example shows a portion of a text-only alert log:

```
Fri Jun 19 17:05:34 2009
Starting ORACLE instance (normal)
LICENSE_MAX_SESSION = 0
LICENSE_SESSIONS_WARNING = 0
Shared memory segment for instance monitoring created
Picked latch-free SCN scheme 2
Autotune of undo retention is turned on.
IMODE=BR
ILAT =12
LICENSE_MAX_USERS = 0
SYS auditing is disabled
Starting up ORACLE RDBMS Version: 11.2.0.0.0.
Using parameter settings in client-side pfile
.
.
.
System parameters with nondefault values:
  db_name           = "my_test"
Fri Jun 19 17:05:37 2009
PMON started with pid=2, OS id=10329
Fri Jun 19 17:05:37 2009
VKTM started with pid=3, OS id=10331 at elevated priority
VKTM running at (20)ms precision
Fri Jun 19 17:05:37 2009
DIAG started with pid=4, OS id=10335
```

As shown in [Example 13-1](#), query V\$DIAG_INFO to locate the alert log.

Trace Files

A **trace file** is an administrative file that contain diagnostic data used to investigate problems. Also, trace files can provide guidance for tuning applications or an instance, as explained in "[Performance Diagnostics and Tuning](#)" on page 18-20.

Types of Trace Files

Each server and background process can periodically write to an associated trace file. The files 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. To enable tracing for a client identifier, service, module, action, session, instance, or database, you must execute the appropriate procedures in the DBMS_MONITOR package or use Oracle Enterprise Manager.

A **dump** is a special type of trace file. Whereas a trace tends to be continuous output of diagnostic data, a dump is typically a one-time output of diagnostic data in response to an event (such as an incident). 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.

See Also:

- "[Session Control Statements](#)" on page 7-8
- *Oracle Database Administrator's Guide* to learn about trace files, dumps, and core files
- *Oracle Database Performance Tuning Guide* to learn about application tracing

Locations of Trace Files

ADR stores trace files in the `trace` subdirectory, as shown in [Figure 13–8](#). 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 map (`.trm`) files. These files contain structural information about trace files and are used for searching and navigation.

See Also: *Oracle Database Administrator's Guide* to learn how to find trace files

Memory Architecture

This chapter discusses the memory architecture of an Oracle 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](#)
- [Overview of the System Global Area](#)
- [Overview of Software Code Areas](#)

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

See Also: [Chapter 15, "Process Architecture"](#)

Basic Memory Structures

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. The SGA is shared by all server and background processes. 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. The PGA is created by Oracle Database when an Oracle process is started.

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)

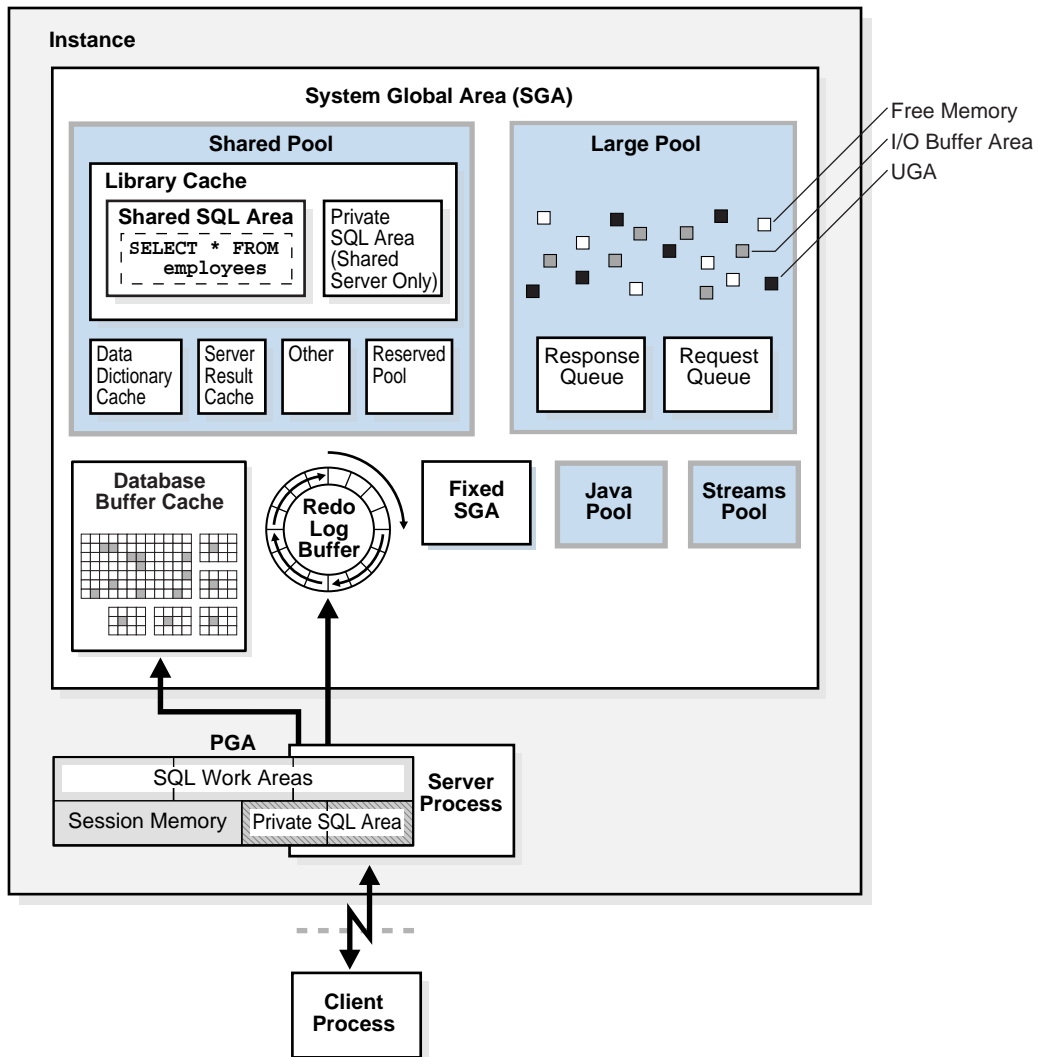
The UGA is memory associated with a user session.

- 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.

Figure 14–1 illustrates the relationships among these memory structures.

Figure 14–1 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 instance memory. The database 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.

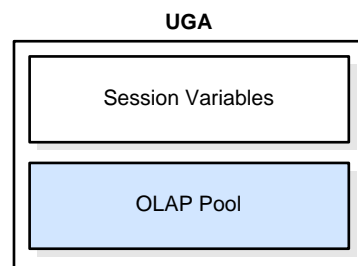
See Also:

- ["Memory Management"](#) on page 18-15 for more information about memory management options for DBAs
- ["Tools for Database Installation and Configuration"](#) to learn about DBCA
- *Oracle Database 2 Day DBA* and *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. [Figure 14–2](#) depicts the UGA.

Figure 14–2 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 (see ["PL/SQL Packages"](#) on page 8-6). 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:

- ["Connections and Sessions"](#) on page 15-4
- *Oracle Database Net Services Administrator's Guide* to learn about shared server connections
- *Oracle OLAP User's Guide* for an overview of Oracle OLAP

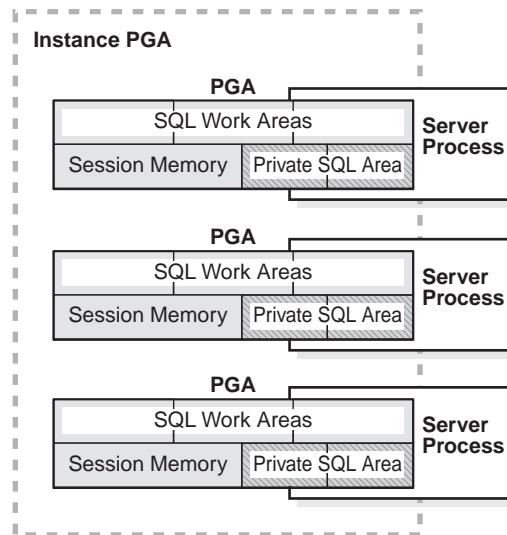
Overview of the Program Global Area

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.

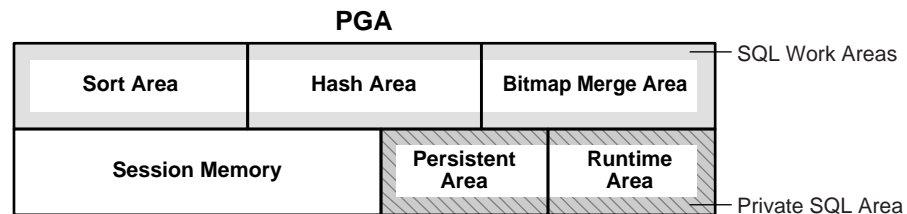
[Figure 14-3](#) shows an instance PGA (collection of all PGAs) for an instance that is not configured for shared servers. You can use an initialization parameter to set a target maximum size of the instance PGA (see ["Summary of Memory Management Methods"](#) on page 18-17). Individual PGAs can grow as needed up to this target size.

Figure 14–3 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. [Figure 14–4](#) shows the possible contents of the PGA for a dedicated server session. Not all of the PGA areas will exist in every case.

Figure 14–4 PGA Contents

Private SQL Area

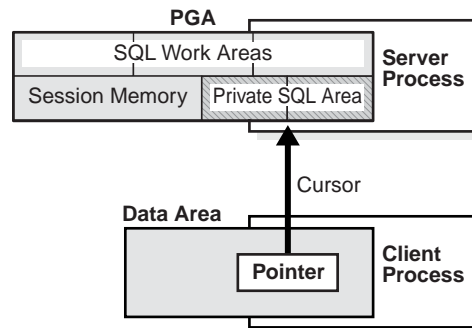
A **private SQL area** holds information about a parsed SQL statement and other session-specific information for processing. When a server process executes SQL or PL/SQL code, the process uses the private SQL area to store **bind variable** values, query execution state information, and query execution work areas.

Do not confuse a *private* SQL area, which is in the UGA, 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 employees` 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 [Figure 14–5](#), 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 14–5 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"](#) on page 14-16
- *Oracle Database Advanced Application Developer's 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.

[Example 14–1](#) shows a **join** of employees and departments with its **query plan**.

Example 14–1 Query Plan for Table Join

```
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 [Example 14–1](#), 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. See "[Memory Management](#)" on page 18-15 for more information.

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. [Table 14–1](#) shows the differences.

Table 14–1 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/DDDL statements	PGA	PGA

See Also: *Oracle Database Net Services Administrator's Guide* to learn how to configure a database for shared server

Overview of the System Global Area

The **SGA** is a read/write memory area that, along with the Oracle background processes, make up a database instance. 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.

Note: The server and background processes do not reside *within* the SGA, but exist in a separate memory space.

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 14-1](#), 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](#)
- [Redo Log Buffer](#)
- [Shared Pool](#)
- [Large Pool](#)
- [Java Pool](#)
- [Streams Pool](#)
- [Fixed SGA](#)

See Also:

- ["Introduction to the Oracle Database Instance"](#) on page 13-1
- *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.

Oracle Database uses the buffer cache to achieve the following goals:

- 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 disk but does not immediately write data blocks to disk. Instead, **database writer (DBWn)** 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 a **flash disk device**, which is a solid state storage device that uses flash memory. The database can improve performance by caching buffers in flash memory instead of reading from magnetic disk.

Note: Database Smart Flash Cache is available only in Solaris and Oracle Enterprise Linux.

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.

The database uses a sophisticated algorithm to make buffer access efficient. Pointers to dirty and nondirty buffers exist on the same **least recently used (LRU) list**, which has a hot end and cold end. A **cold buffer** is one that has not been recently used. A **hot buffer** is frequently accessed and has been recently used.

Note: Conceptually, there is only one LRU, but for **concurrency** the database actually uses several LRUs.

Buffer Modes

When a client requests data, Oracle Database retrieves buffers from the database buffer cache in either of the following modes:

- 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"](#) on page 9-3
- *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, and then a logical I/O to read the cached buffer.

Buffer Writes The **database writer (DBWn)** process periodically writes cold, dirty buffers to disk. DBWn 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 DBWn 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**. DBWn 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 \(DBWn\)"](#) on page 15-8
- *Oracle Database Performance Tuning Guide* to learn how to diagnose and tune buffer write issues

Buffer Reads When the number of clean or 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

DBWn 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.

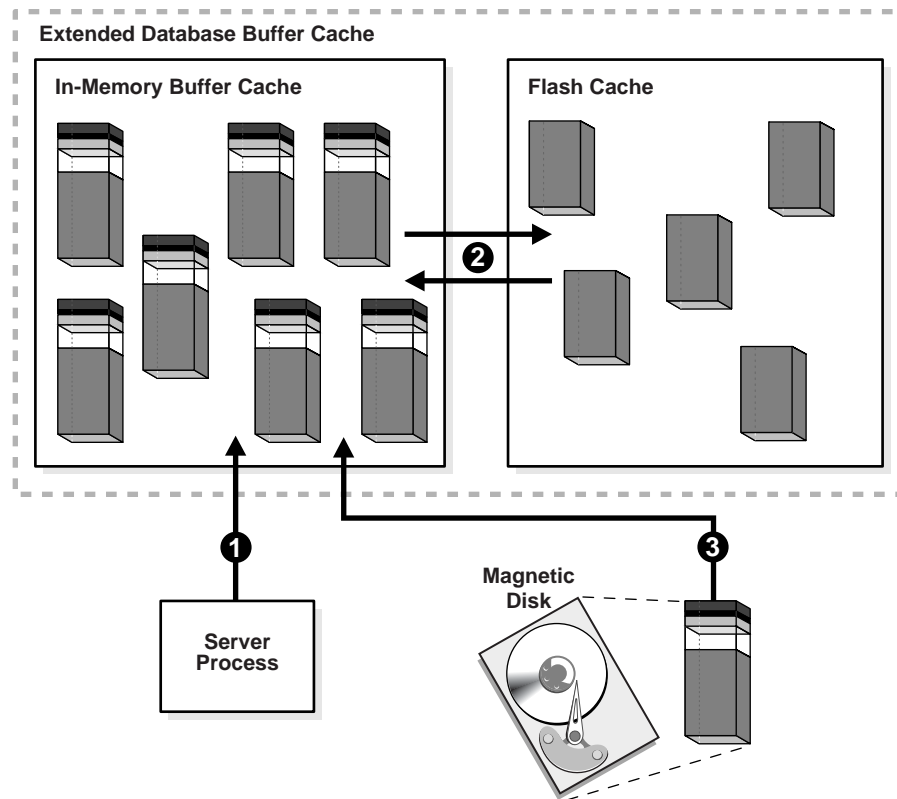
If the process finds the buffer header, then the database performs an **optimized physical read** of the buffer body from the flash cache into the in-memory cache.

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 into memory (a **physical read**)

- b. Performs a logical read of the buffer that was read into memory

Figure 14–6 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 14–6 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.

Buffers and Full Table Scans 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** (see "Segment Space and the High Water Mark" on page 12-27). 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.

Blocks read into the database cache as the result of a full scan of a large table are treated differently from other types of reads. The blocks are immediately available for reuse to prevent the scan from effectively cleaning out the buffer cache.

In the rare case where the default behavior is not desired, you can change the `CACHE` attribute of the table. In this case, the database does not force or pin the blocks in the buffer cache, but ages them out of the cache in the same way as any other 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.

See Also:

- *Oracle Database SQL Language Reference* for information about the `CACHE` clause
- *Oracle Database Performance Tuning Guide* to learn how to interpret buffer cache advisory statistics

Buffer Pools

A **buffer pool** is a collection of buffers. The database buffer cache is divided into one or more buffer pools.

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.

The possible buffer pools are as follows:

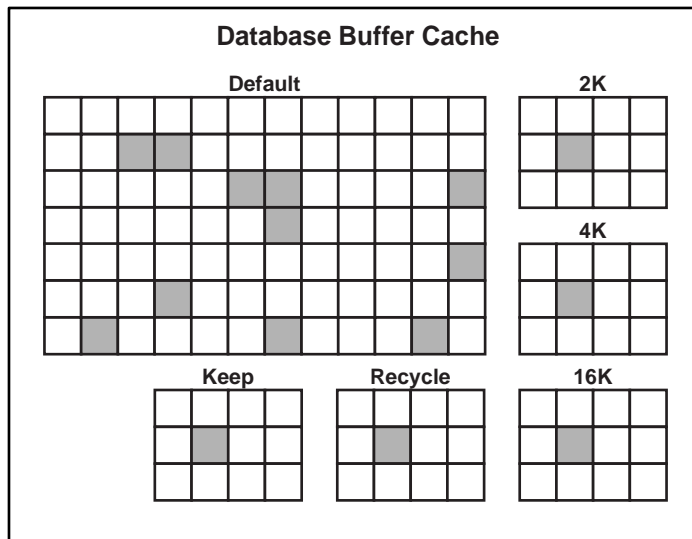
- **Default pool**
This pool is the location where blocks are normally cached. Unless you manually configure separate pools, the default pool is the only buffer pool.
- **Keep pool**
This pool is intended for blocks that were accessed frequently, but which aged out of the default pool because of lack of space. The goal of the keep buffer pool is to retain objects in memory, thus avoiding I/O operations.
- **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 (see "Database Block Size" on page 12-7). 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.

Figure 14-7 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 14–7 Database Buffer Cache



See Also:

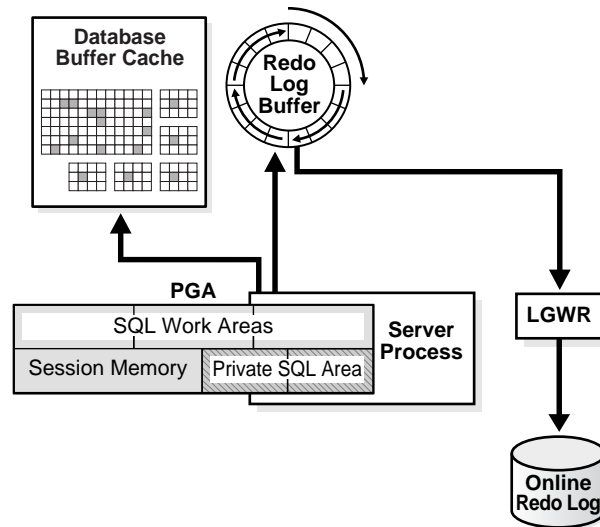
- *Oracle Database 2 Day DBA and Oracle Database Administrator's Guide* to learn more about buffer pools
- *Oracle Database Performance Tuning Guide* to learn how to use multiple buffer pools

Redo Log Buffer

The **redo log buffer** is a circular buffer in the SGA that stores redo entries describing changes made to the database. **Redo entries** contain 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.

Oracle 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 (LGWR)** writes the redo log buffer to the active online redo log group on disk. [Figure 14–8](#) shows this redo buffer activity.

Figure 14–8 Redo Log Buffer



LGWR writes redo sequentially to disk while DBW_n 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_n 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.

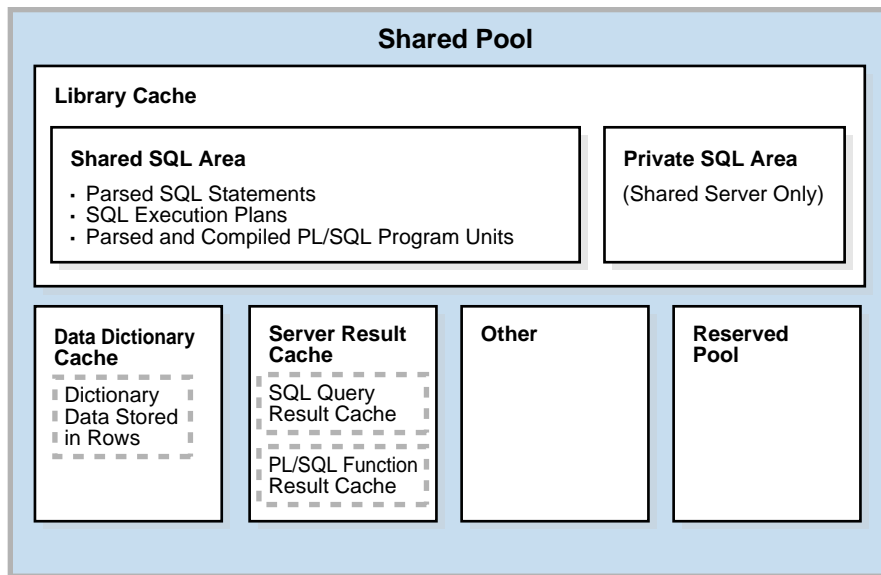
See Also:

- ["Log Writer Process \(LGWR\)"](#) on page 15-9 and ["Importance of Checkpoints for Instance Recovery"](#) on page 13-13
- *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 14–9](#).

Figure 14–9 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. 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 following SQL areas:

- Shared 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.
- Private SQL area

Each session issuing a SQL statement has a private SQL area in its PGA (see "[Private SQL Area](#)" on page 14-5). 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**.

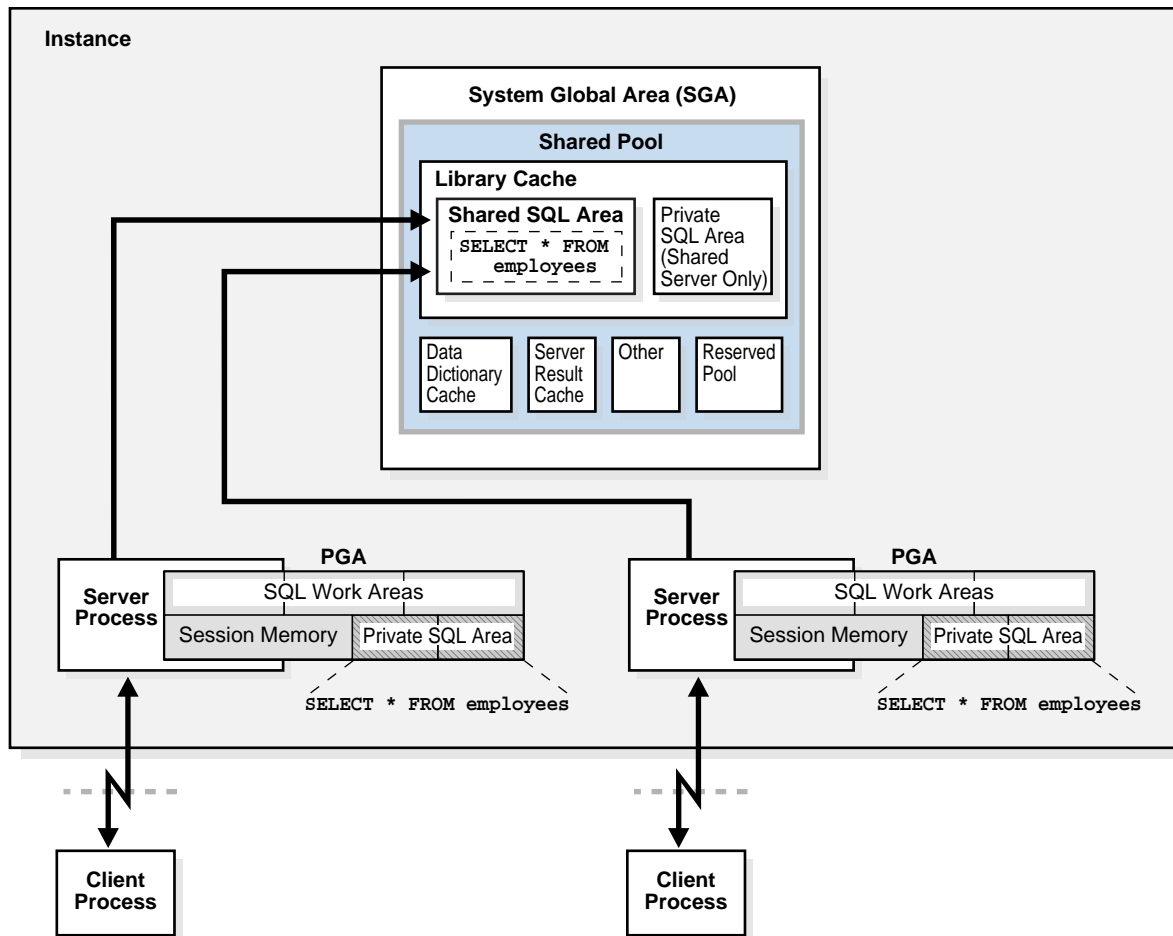
In either case, the private SQL area for the user points to the shared SQL area that contains the statement and execution plan.

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 14-10](#) 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 14–10 Private SQL Areas and Shared SQL Area

**See Also:**

- *Oracle Database Performance Tuning Guide* to learn more about managing the library cache
- *Oracle Database Advanced Application Developer's 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. The memory size depends on the complexity of the statement.

In general, an item in the shared pool stays until it is removed according to an 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 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 for infrequently used items. A shared SQL area can be removed from the shared pool even if the shared SQL area corresponds to an open cursor that has not been used for some time. If the open cursor is subsequently used to run its statement, then Oracle Database reparses the statement and allocates a new shared SQL area.

The database also removes a shared SQL area from the shared pool in the following circumstances:

- If statistics are gathered for a table, **table cluster**, or index, then by default the database gradually removes all shared SQL areas that contain statements referencing the analyzed object after a period of time. The next time a removed statement is run, the database parses it in a new shared SQL area to reflect the new statistics for the schema object.
- If a schema object is referenced in a SQL statement, and if this object is later modified by a DDL statement, then the database invalidates the shared SQL area. The optimizer must reparse the statement the next time it is run.
- If you change the global database name, then the database removes all information from the shared pool.

You can use the `ALTER SYSTEM FLUSH SHARED_POOL` statement to manually remove all information in the shared pool to assess the performance that can be expected after an instance restart.

See Also:

- *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:

- [Chapter 6, "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

Unlike the buffer pools, the **server result cache** holds result sets and not data blocks. The server result cache contains the **SQL query result cache** and **PL/SQL function result cache**, which share the same infrastructure.

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.

See Also:

- *Oracle Database Administrator's Guide* for information about sizing 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

SQL Query Result Cache The database can store the results of queries and query fragments in the **SQL query result cache**, using the cached results for future queries and query fragments. Most applications benefit from this performance improvement.

For example, suppose an application runs the same `SELECT` statement repeatedly. If the results are cached, then the database returns them immediately. In this way, the database avoids the expensive operation of rereading blocks and recomputing results. The database automatically invalidates a cached result whenever a transaction modifies the data or metadata of database objects used to construct that cached result.

Users can annotate a query or query fragment with a `RESULT_CACHE` **hint** to indicate that the database should store results in the SQL query result cache. The `RESULT_CACHE_MODE` initialization parameter determines whether the SQL query result cache is used for all queries (when possible) or only for annotated queries.

See Also:

- *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

PL/SQL Function Result Cache The **PL/SQL function result cache** stores function result sets. Without caching, 1000 calls of a 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 cached 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 that are used to compute a cached result so that if any of them are updated, the cached result becomes invalid and must be recomputed.

The cache can accumulate many results—one result for every unique combination of parameter values with which each result-cached function was invoked. If the database needs more memory, then it ages out one or more cached results.

See Also:

- *Oracle Database Advanced Application Developer's 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

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.

Allocation of memory from the shared pool is performed 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

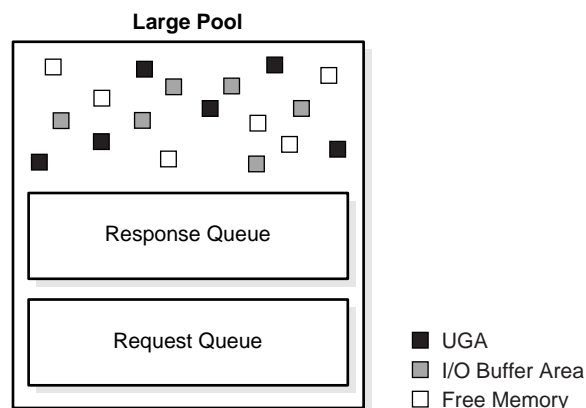
Large Pool

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 the parallel execution of statements
- Buffers for Recovery Manager (RMAN) I/O slaves

By allocating session memory from the large pool for shared SQL, the database avoids performance overhead caused by shrinking the shared SQL cache. By allocating memory in large buffers for RMAN operations, I/O server processes, and parallel buffers, the large pool can satisfy large memory requests better than the shared pool.

[Figure 14–11](#) is a graphical depiction of the large pool.

Figure 14–11 Large Pool

The large pool is different from reserved space in the shared pool, which uses the same LRU list as other memory allocated from the shared pool. The large pool does not have an LRU list. Pieces of memory are allocated and cannot be freed until they are done being used. As soon as a chunk of memory is freed, other processes can use it.

See Also:

- ["Dispatcher Request and Response Queues"](#) on page 16-12 to learn about allocating session memory for shared server
- *Oracle Database Advanced Application Developer's Guide* to learn about Oracle XA
- *Oracle Database Performance Tuning Guide* for more information about the large pool
- ["Parallel Execution"](#) on page 15-15 for information about allocating memory for parallel execution

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

Streams Pool

The **Streams pool** stores buffered queue messages and provides memory for Oracle Streams capture processes and apply processes. The Streams pool is used exclusively by Oracle Streams.

Unless you specifically configure it, the size of the Streams pool starts at zero. The pool size grows dynamically as required by Oracle Streams.

See Also: *Oracle Database 2 Day + Data Replication and Integration Guide* and *Oracle Streams Replication Administrator's Guide*

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** (see "[Overview of Automatic Locks](#)" on page 9-17)

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.

Overview of Software Code Areas

Software code areas are portions of memory that store 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 Windows. See your operating system-specific documentation for more information.

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. The mechanism depends on the operating system. For example, on Linux an Oracle background process is a Linux process. On Windows, an Oracle background process is a thread of execution within a process.

Code modules are run by processes. 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** (see "[Trace Files](#)" on page 13-22).

See Also: "[Tools for Database Administrators](#)" on page 18-2 and "[Tools for Database Developers](#)" on page 19-1

Multiple-Process Oracle Database Systems

Multiple-process Oracle (also called **multiuser Oracle**) uses several processes to run different parts of the Oracle Database code and additional 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.

Types of Processes

A database instance contains or interacts with the following types of processes:

- **Client processes** run the application or Oracle tool code.
- **Oracle processes** run the Oracle database code. Oracle processes including the following subtypes:
 - **Background processes** start with the database instance and perform maintenance tasks such as performing **instance recovery**, cleaning up processes, writing redo buffers to disk, and so on.
 - **Server processes** perform 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.

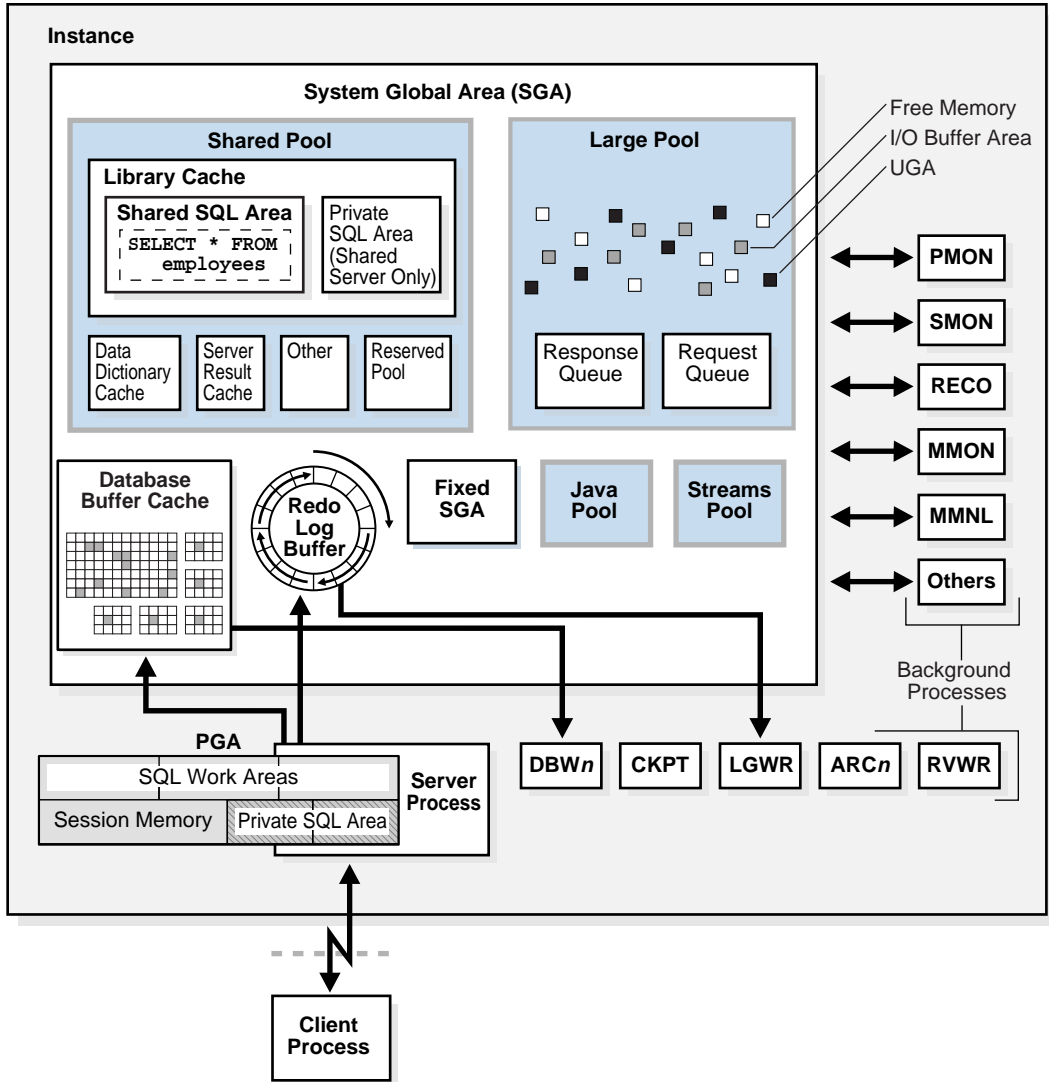
Note: Server processes, and the process memory allocated in these processes, run in the instance. The instance continues to function when server processes terminate.

- **Slave processes** perform 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, the code for connected users can be configured 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 15–1 shows a **system global area (SGA)** and background processes using dedicated server connections. For each user connection, the application is run by a **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 15-1 Oracle Processes and the SGA



See Also:

- ["Dedicated Server Architecture"](#) on page 16-9 and ["Shared Server Architecture"](#) on page 16-11
- Your Oracle Database operating system-specific documentation for more details on configuration choices
- *Oracle Database Reference* to learn about the V\$PROCESS view

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. 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 connects over the network to database `sample` on a different host (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)
```

Connections and Sessions

A **connection** is a physical communication pathway between a client process and a database instance. 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).

A **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.

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 [Figure 15-2](#), 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 15–2 One Session for Each Connection

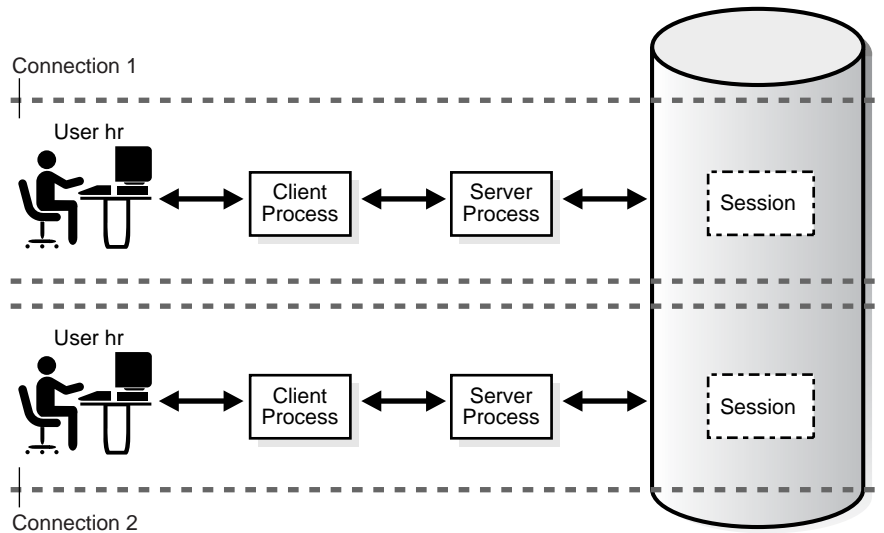
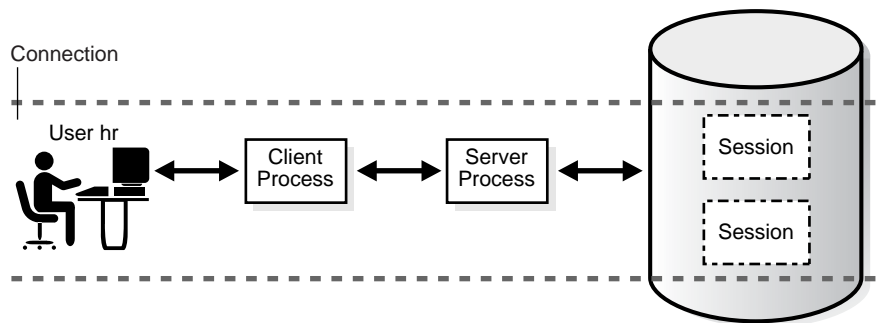


Figure 15–3 illustrates a case in which user hr has a single connection to a database, but this connection has two sessions.

Figure 15–3 Two Sessions in One Connection



Generating an autotrace report of SQL statement execution statistics re-creates the scenario in Figure 15–3. Example 15–2 connects SQL*Plus to the database as user SYSTEM and enables tracing, thus creating a new session (sample output included).

Example 15–1 One Connection with Two Sessions

```
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 in [Example 15-1](#) actually ends the *sessions*, not the connection. Opening a new terminal and connecting to the instance as a different user, the query in [Example 15-2](#) shows that the connection from [Example 15-1](#) is still active.

Example 15-2 Connection with No Sessions

```
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"](#) on page 16-11

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** (see ["Stages of SQL Processing"](#) on page 7-15)
- Execute PL/SQL code
- Read data blocks from data files into the database buffer cache (the `DBWn` 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 (see ["Dedicated Server Architecture"](#) on page 16-9). 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 ["PGA Usage in Dedicated and Shared Server Modes"](#) on page 14-7).

Shared Server Processes

In shared server connections, client applications connect over a network to a **dispatcher process**, not a server process (see ["Shared Server Architecture"](#) on page 16-11). 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** (see ["Large Pool"](#) on page 14-21). The first available shared server process takes the request from the queue and processes it. Afterward, the shared server place 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.

Overview of Background Processes

A multiprocess Oracle database uses some additional processes called **background processes**. 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 log file is ready to be archived, LGWR signals another process to archive the 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](#)

See Also: *Oracle Database Reference* for descriptions of all the background processes

Mandatory Background Processes

The **mandatory background processes** are present in all typical database configurations. These processes run by default in a database instance started with a minimally configured initialization parameter file (see [Example 13-1](#) on page 13-20).

This section describes the following mandatory background processes:

- [Process Monitor Process \(PMON\)](#)
- [System Monitor Process \(SMON\)](#)
- [Database Writer Process \(DBWn\)](#)
- [Log Writer Process \(LGWR\)](#)
- [Checkpoint Process \(CKPT\)](#)
- [Manageability Monitor Processes \(MMON and MMNL\)](#)
- [Recoverer Process \(RECO\)](#)

See Also:

- *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)

The **process monitor (PMON)** monitors the other background processes and performs process recovery when a server or dispatcher process terminates abnormally. PMON is responsible for cleaning up the database buffer cache and freeing resources that the client process was using. For example, PMON resets the status of the active **transaction table**, releases **locks** that are no longer required, and removes the process ID from the list of active processes.

PMON also registers information about the instance and dispatcher processes with the **Oracle Net listener** (see "[The Oracle Net Listener](#)" on page 16-6). When an instance starts, PMON polls the listener to determine whether it is running. If the listener is running, then PMON passes it relevant parameters. If it is not running, then PMON periodically attempts to contact it.

System Monitor Process (SMON)

The **system monitor process (SMON)** is in charge of a variety of system-level cleanup duties. The 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 (DBWn)

The **database writer process (DBWn)** writes the contents of database buffers to data files. DBWn processes write modified buffers in the database buffer cache to disk (see "[Database Buffer Cache](#)" on page 14-9).

Although one database writer process (DBW0) is adequate for most systems, you can configure additional processes—DBW1 through DBW9 and DBWa through DBWj—to improve write performance if your system modifies data heavily. These additional DBWn processes are not useful on uniprocessor systems.

The DBWn 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 DBWn to write. DBWn writes dirty buffers to disk asynchronously if possible while performing other processing.
- DBWn periodically writes buffers to advance the **checkpoint**, which is the position in the **redo thread** from which instance recovery begins (see "[Overview of Checkpoints](#)" on page 13-11). The log position of the checkpoint is determined by the oldest dirty buffer in the buffer cache.

In many cases the blocks that DBWn writes are scattered throughout the disk. Thus, the writes tend to be slower than the sequential writes performed by LGWR. DBWn performs multiblock writes when possible to improve efficiency. The number of blocks written in a multiblock write varies by operating system.

See Also: *Oracle Database Performance Tuning Guide* for advice on configuring, monitoring, and tuning DBWn

Log Writer Process (LGWR)

The **log writer process (LGWR)** manages the redo log buffer. LGWR writes one contiguous 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 (see "[Committing Transactions](#)" on page 10-10).
- 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.
- DBWn must write modified buffers to disk.

Before DBWn can write a dirty buffer, redo records associated with changes to the buffer must be written to disk (the **write-ahead protocol**). If DBWn finds 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. 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 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 DBWn to write them to the data files.

Note: LGWR can write redo log entries to disk before a transaction commits. 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.

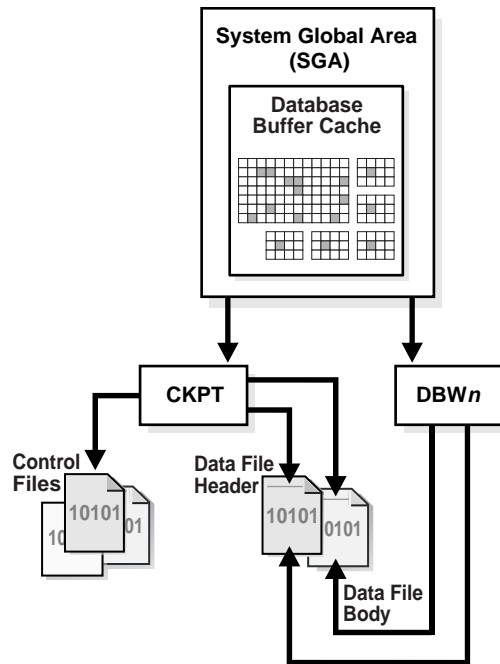
See Also:

- ["How Oracle Database Writes to the Online Redo Log"](#) on page 11-12 and ["Redo Log Buffer"](#) on page 14-14
- *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 DBWn to write blocks to disk. Checkpoint information includes the checkpoint position, SCN, location in online redo log to begin recovery, and so on. As shown in [Figure 15-4](#), CKPT does not write data blocks to data files or redo blocks to online redo log files.

Figure 15–4 Checkpoint Process



See Also: ["Overview of Checkpoints"](#) on page 13-11

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: ["Automatic Workload Repository \(AWR\)"](#) on page 18-21 and ["Active Session History \(ASH\)"](#) on page 18-23

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 Streams Advanced Queuing (AQ)

or **Oracle Automatic Storage Management (Oracle ASM)** are only available when these features are enabled.

This section describes some common optional processes:

- [Archiver Processes \(ARCn\)](#)
- [Job Queue Processes \(CJQ0 and Jnnn\)](#)
- [Flashback Data Archiver Process \(FBDA\)](#)
- [Space Management Coordinator Process \(SMCO\)](#)

See Also:

- ["Oracle Streams Advanced Queuing \(AQ\)"](#) on page 17-23
- *Oracle Database Reference* for descriptions of background processes specific to AQ and ASM

Archiver Processes (ARCn)

The **archiver processes (ARCn)** copy 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. ARCn processes exist *only* when the database is in **ARCHIVELOG mode** and automatic archiving is enabled.

See Also:

- ["Archived Redo Log Files"](#) on page 11-15
- *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)

Oracle Database uses **job queue processes** to run 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 a large number of 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 (see ["Oracle Scheduler"](#) on page 18-19). 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 jobs
- *Oracle Streams Advanced Queuing User's Guide* to learn about AQ background processes

Flashback Data Archiver Process (FBDA)

The **flashback data archiver 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 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 far the archiving of tracked transactions has occurred.

Space Management Coordinator Process (SMCO)

The SMCO process coordinates the execution of various space management related tasks, such as proactive space allocation and space reclamation. SMCO dynamically spawns slave processes (*Wnnn*) to implement the task.

See Also: *Oracle Database Advanced Application Developer's Guide* to learn about Flashback Data Archive

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.

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 make use of 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. DBWR 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 Query Slaves

In **parallel execution** or **parallel processing**, 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.

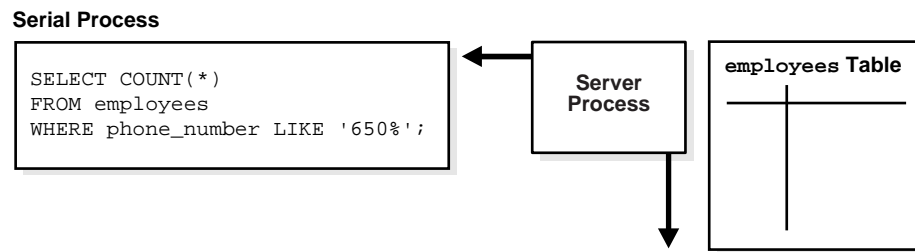
Parallel execution reduces response time for data-intensive operations on large databases such as data warehouses. Symmetric multiprocessing (SMP) and clustered system gain the largest performance benefits from parallel execution because statement processing can be split up among multiple CPUs. Parallel execution can also benefit certain types of **OLTP** and hybrid systems.

In Oracle RAC systems, the service placement of a particular service controls parallel execution. Specifically, parallel processes run on the nodes on which you have configured the service. By default, Oracle Database runs the parallel process only on the 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.

See Also:

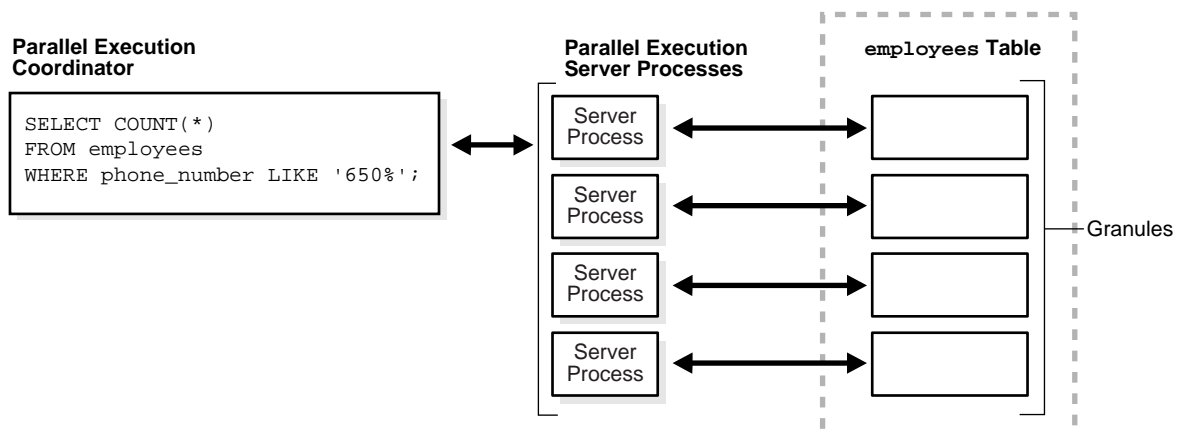
- *Oracle Database Data Warehousing Guide* and *Oracle Database VLDB and Partitioning Guide* to learn more about parallel execution
- *Oracle Real Application Clusters Administration and Deployment Guide* for considerations regarding parallel execution in Oracle RAC environments

Serial Execution In **serial execution**, 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 15-5](#).

Figure 15–5 Serial Full Table Scan

Parallel Execution In **parallel execution**, the server process acts as the **parallel execution coordinator** responsible for parsing the query, allocating and controlling the slave processes, and sending output to the user. Given a **query plan** for a SQL 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 slave processes executing the operators.

Figure 15–6 shows a parallel scan of the employees table. The table is divided dynamically (**dynamic partitioning**) into load units called **granules**. Each granule is a range of data blocks of the table read by a single slave process, called a **parallel execution server**, which uses *Pnnn* as a name format.

Figure 15–6 Parallel Full Table Scan

The database maps granules to execution servers at execution time. When an execution server finishes reading the rows corresponding to a granule, and when granules remain, it obtains another granule from the 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**.

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.

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

Application and Networking 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 Networking 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 Client/Server Architecture

In the Oracle Database environment, the database application and the database are separated into a **client/server architecture**:

- 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**. Front-end and back-end processing occurs on different computers. In [Figure 16-1](#), the client and server are located on different hosts connected through Oracle Net Services.

Figure 16–1 Client/Server Architecture and Distributed Processing

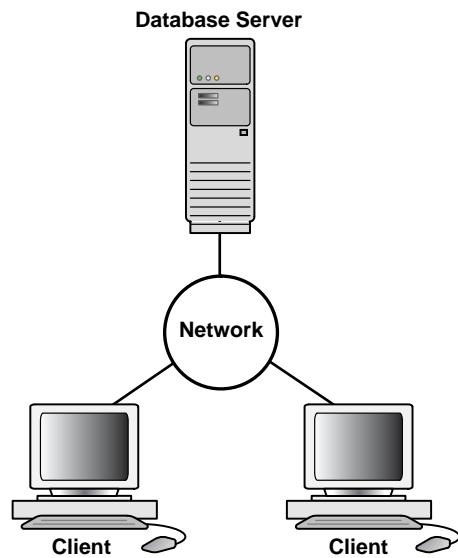
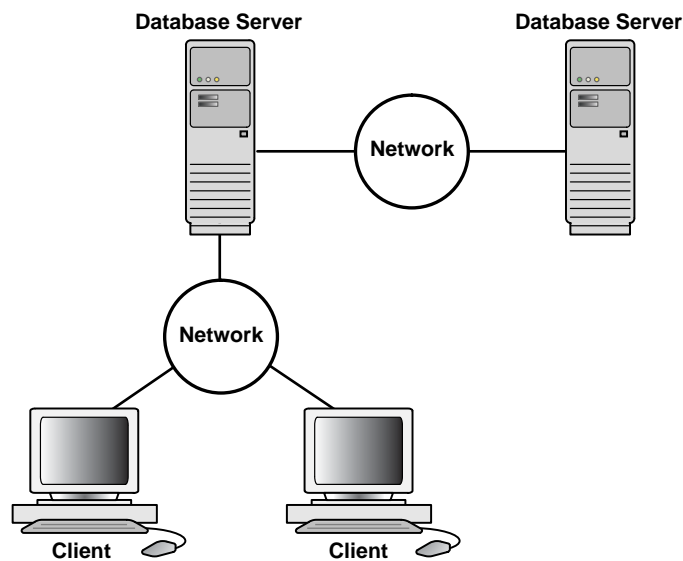


Figure 16–2 is a variation that depicts a **distributed database**. In this example, a database on one host accesses data on a separate database located on a different host.

Figure 16–2 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 the following benefits:

- 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. As a result, 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 be **scaled** 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.

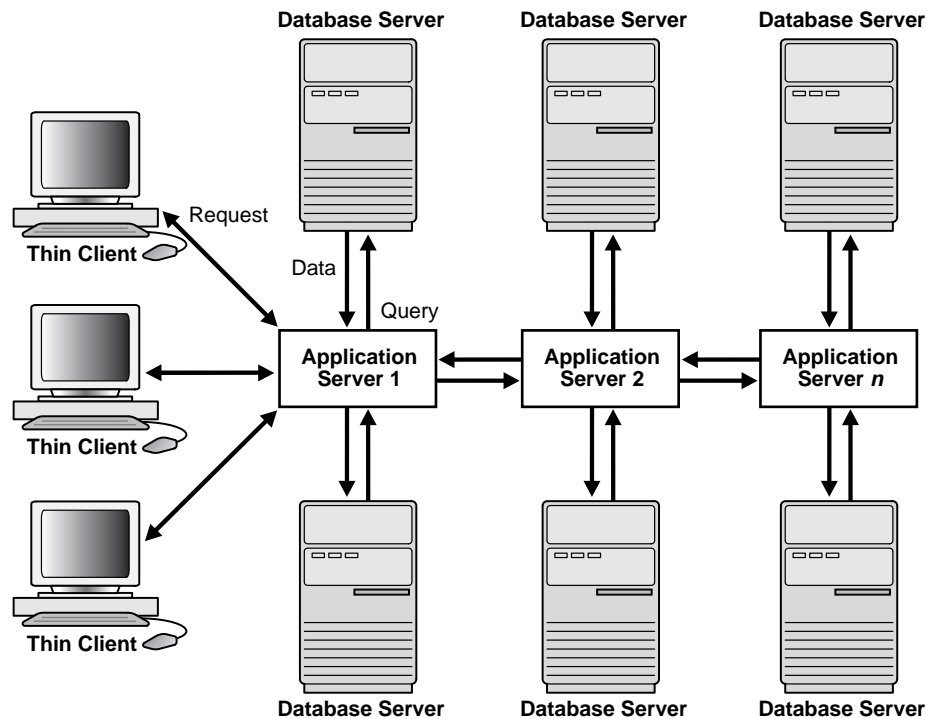
See Also: *Oracle Database Administrator's Guide* to learn more about distributed databases

Overview of Multitier Architecture

In a traditional multitier architecture environment, 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 16-3](#).

Figure 16–3 A Multitier Architecture Environment

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 some 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 that client. The privileges of the application server should be restricted 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 all of the **query** processing.

The database server can audit operations performed by the application server on behalf of clients and operations performed by the application server on its own behalf (see "**Monitoring**" on page 17-5). 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.

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. The services can be dynamically discovered and queried on available functions and calling sequences.

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 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 (see "[PL/SQL Subprograms](#)" on page 8-3)
- 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:

- *Oracle XML DB Developer'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: "[Overview of Grid Computing](#)" on page 17-11 for more detailed information about server and storage grids

Overview of Oracle Networking 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 Oracle Database **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.

See Also: *Oracle Database Net Services Administrator's Guide* for an overview of Oracle Net architecture

How Oracle Net Services Works

Oracle Database protocols take SQL statements from the interface of the Oracle applications and package them for transmission to Oracle Database 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**, also called 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 PMON 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.

[Figure 16–4](#) shows two databases, each on a separate host. The database environment is serviced by two listeners, each on a separate host. The PMON process running in each database instance communicates with both listeners to register the database.

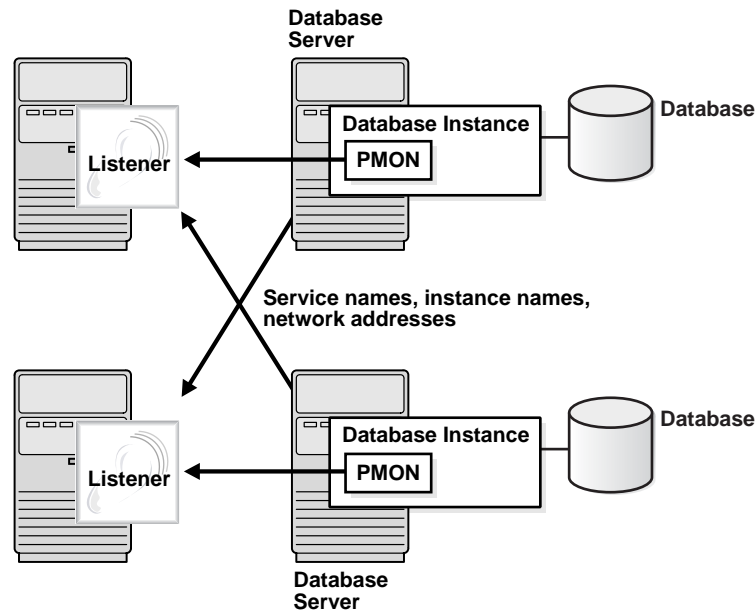
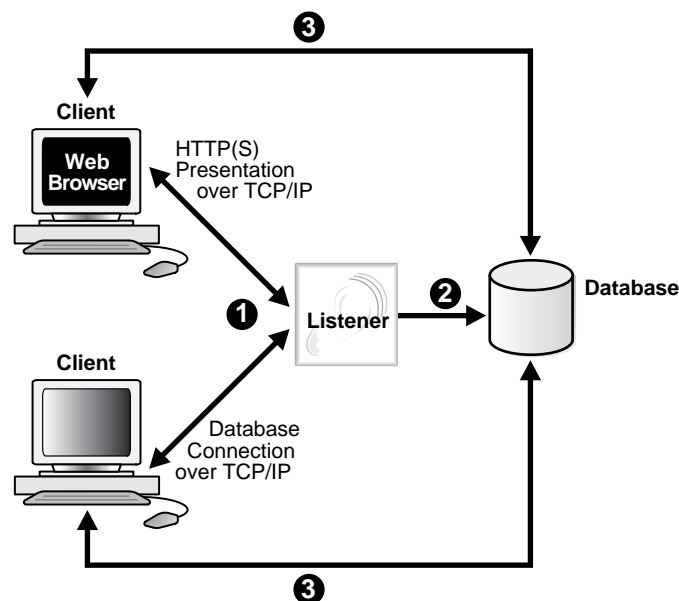


Figure 16-4 shows a browser making an HTTP connection and a client making a database connection through a listener. The listener does not need to reside on the database host.

Figure 16-4 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.

See Also: ["Overview of Client Processes"](#) on page 15-3 and ["Overview of Server Processes"](#) on page 15-6

Service Names

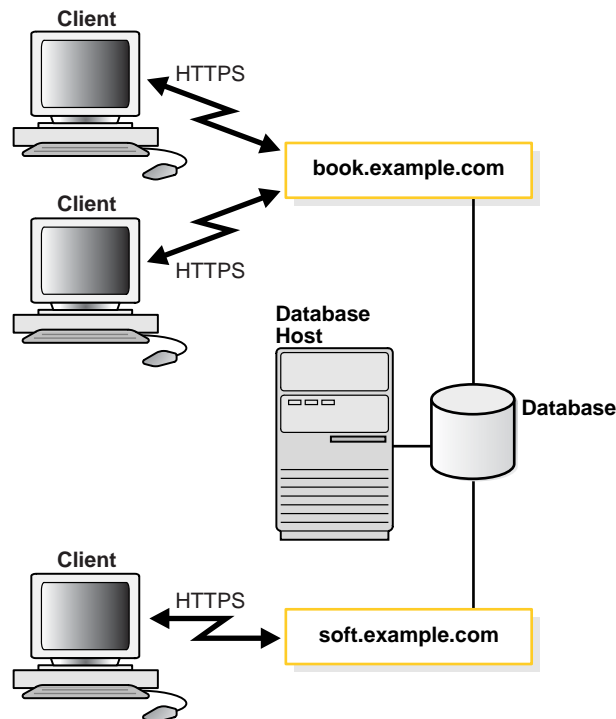
In the context of net services, a **service** is a set of one or more database instances. A **service name** is a logical representation of a service used for client connections.

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 16-5](#) 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 16-5 Multiple Services Associated with One Database



See Also: *Oracle Database Net Services Administrator's Guide* to learn more about naming methods

Service Registration

Service registration is a feature by which the PMON process dynamically registers instance information with a listener, which enables the listener to forward client connection requests to the appropriate service handler. PMON 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.

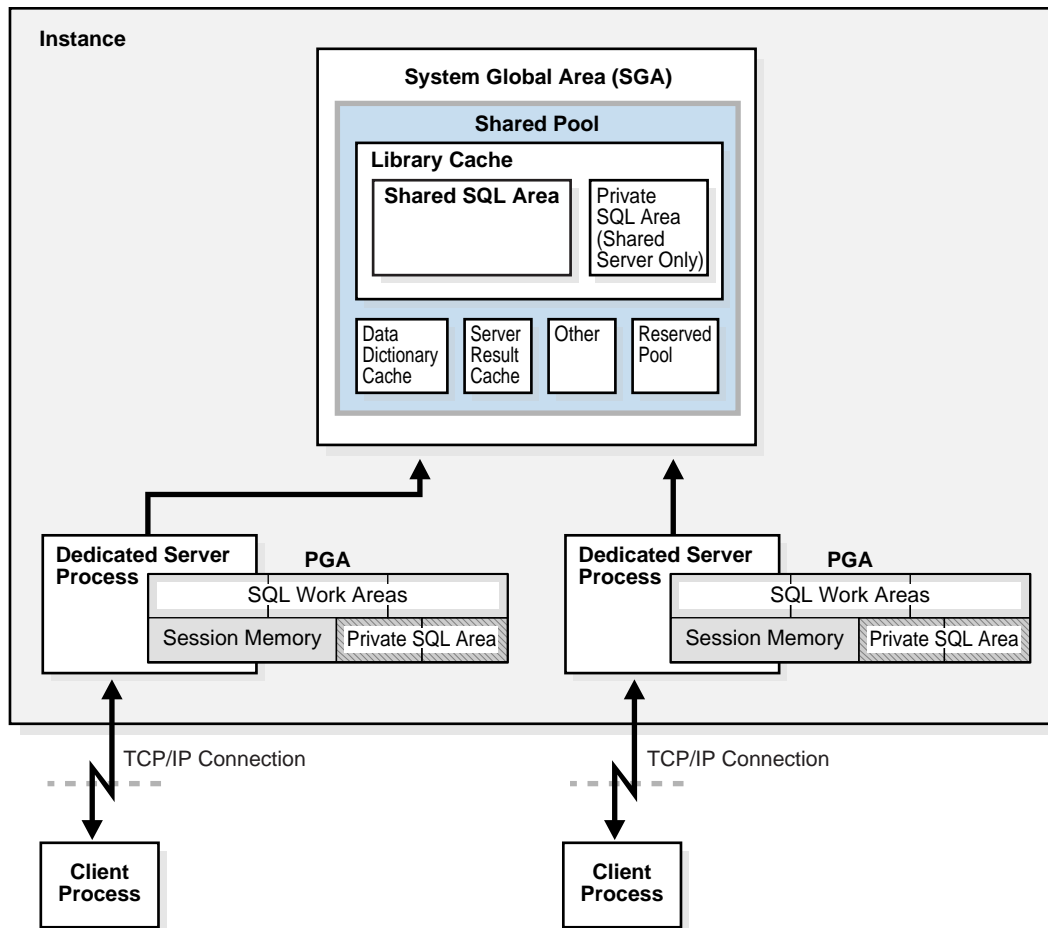
The initialization parameter `SERVICE_NAMES` lists the services an instance belongs to. On startup, each instance registers with the listeners of other instances belonging to the same services. During database operations, the instances of each service pass information about CPU use and current connection counts to all listeners in the same services. This communication enables dynamic load balancing and connection failover.

See Also:

- ["Process Monitor Process \(PMON\)" on page 15-8](#)
- *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). This server process is separate from the client process and acts only on its behalf, as shown in [Figure 16-6](#).

Figure 16–6 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 16–6 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, eliminating 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 (Dnnn)**
- One or more shared server processes

Note that 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.

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 (see "[Large Pool](#)" on page 14-21).

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.

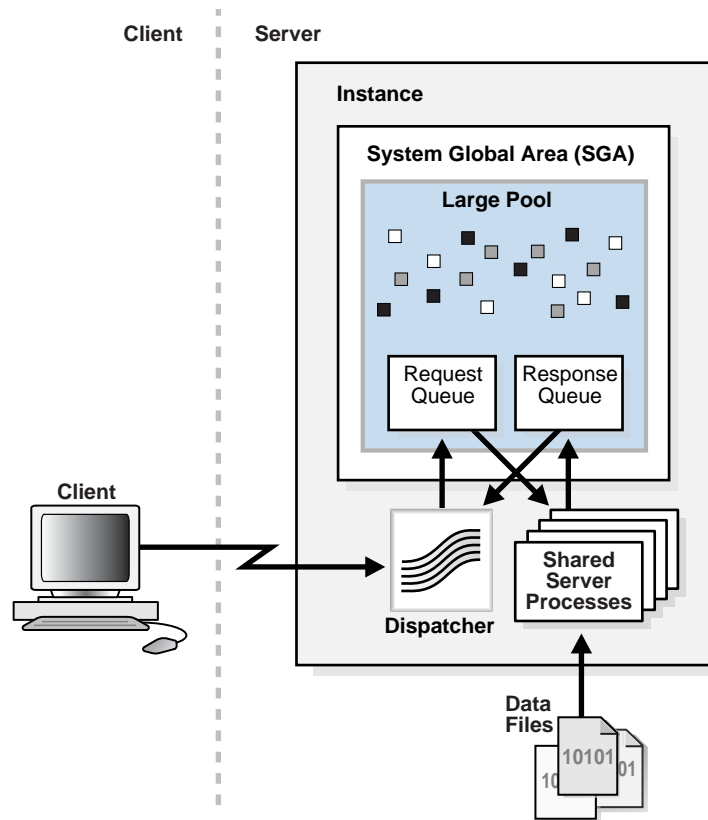
A different server process can handle each database call. Therefore, requests to parse a query, fetch the first row, fetch the next row, and close the result set may each be processed by a different shared server.

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 16-7](#) shows how client processes communicate with the dispatcher across the API and how the dispatcher communicates user requests to shared server processes.

Figure 16–7 The Shared Server Configuration and Processes



Dispatcher Processes (Dnnn) 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 and establishes an appropriate connection.

See Also: *Oracle Database Net Services Administrator's Guide* to learn how to configure dispatchers

Shared Server Processes (Snnn) 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 (see "[Overview of the Program Global Area](#)" on page 14-4). 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.

Restricted Operations of the Shared Server

Certain 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 want to use 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:

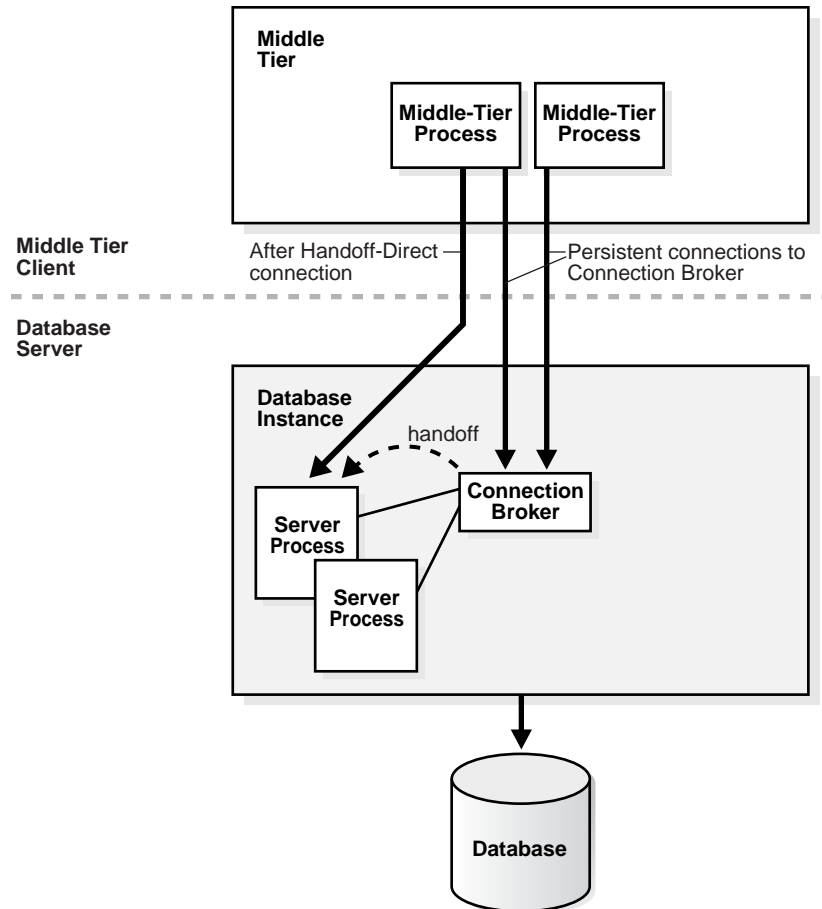
- Complements middle-tier connection pools that share connections between threads in a middle-tier process.
- Enables database connections to be shared across multiple middle-tier processes. These middle-tier processes may belong to the same or different middle-tier host.
- Enables a significant reduction 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 16–8](#), 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 16–8 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"](#) on page 15-4
- *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 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 the following pieces:

- 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

Drivers are pieces of software that transport data, usually across a network. They perform operations such as connect, disconnect, signal errors, and test for errors. Drivers are specific to a communications protocol.

There is always a default driver. 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 is provided by the host operating system. 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.

Part VI

Oracle Database Administration and Development

This part describes summarizes topics that are essential for database administrators and developers.

This part contains the following chapters:

- [Chapter 17, "Topics for Database Administrators and Developers"](#)
- [Chapter 18, "Concepts for Database Administrators"](#)
- [Chapter 19, "Concepts for Database Developers"](#)

Topics for Database Administrators and Developers

The previous parts of this manual described the basic architecture of Oracle Database. This chapter summarizes common database topics that are important for both database administrators and developers, and provides pointers to other manuals, not an exhaustive account of database features.

This chapter contains the following sections:

- [Overview of Database Security](#)
- [Overview of High Availability](#)
- [Overview of Grid Computing](#)
- [Overview of Data Warehousing and Business Intelligence](#)
- [Overview of Oracle Information Integration](#)

See Also: [Chapter 18, "Concepts for Database Administrators"](#) discusses topics specific to DBAs. [Chapter 19, "Concepts for Database Developers"](#) discusses topics for developers.

Overview of Database Security

In general, **database security** involves user authentication, encryption, access control, and monitoring.

User Accounts

Each Oracle database has a list of valid database **users**. The database contains several default accounts, including the default administrative account `SYSTEM` (see ["SYS and SYSTEM Schemas"](#) on page 2-5). You can create user accounts as needed.

To access a database, a user must provide a valid user name and authentication credential. The credential may be a password, Kerberos ticket, or public key infrastructure (PKI) certificate. You can configure database security to lock accounts based on failed login attempts.

Privilege and Role Authorization

In general, **database access control** involves restricting data access and database activities. For example, you can restrict users from querying specified tables or executing specified database commands.

A user **privilege** is the right to run specific SQL statements. Privileges can be divided into the following categories:

- System privilege

This is the right to perform a specific action in the database, or perform an action on any objects of a specific type. For example, `CREATE USER` and `CREATE SESSION` are system privileges.
- Object privilege

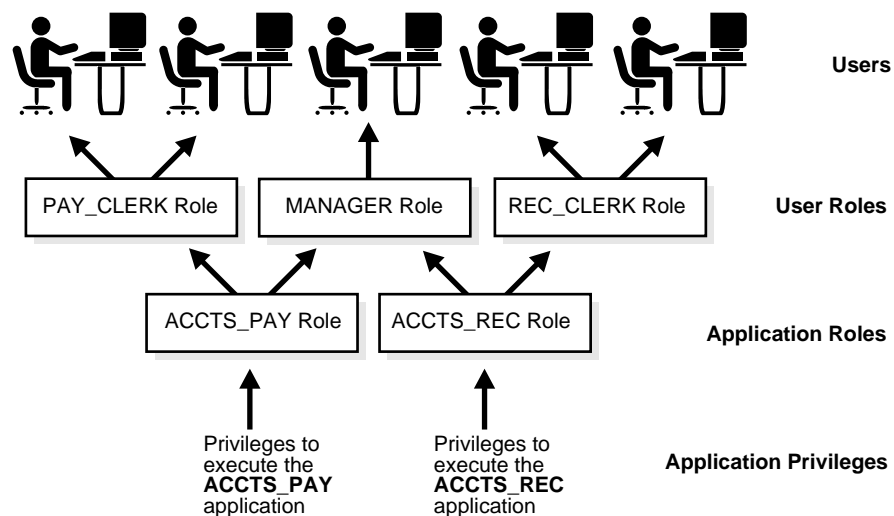
This is the right to perform a specific action on an object, for example, query the `employees` table. Privilege types are defined by the database.

Privileges are granted to users at the discretion of other users. Administrators should grant privileges to users so they can accomplish tasks required for their jobs. Good security practice involves granting a privilege only to a user who requires that privilege to accomplish the necessary work.

A **role** is a named group of related privileges that you grant to users or other roles. A role helps manage privileges for a database application or user group.

Figure 17–1 depicts a common use for roles. The roles `PAY_CLERK`, `MANAGER`, and `REC_CLERK` are assigned to different users. The application role `ACCTS_PAY`, which includes the privilege to execute the `ACCTS_PAY` application, is assigned to users with the `PAY_CLERK` and `MANAGER` role. The application role `ACCTS_REC`, which includes the privilege to execute the `ACCTS_REC` application, is assigned to users with the `REC_CLERK` and `MANAGER` role.

Figure 17–1 Common Uses for Roles



See Also:

- *Oracle Database 2 Day + Security Guide* and *Oracle Database Security Guide* to learn how to manage privileges
- *Oracle Database Security Guide* to learn about using roles for security
- *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn how to administer roles
- *Oracle Database Reference* to learn about the `SESSION_PRIVS` view

Profiles

In the context of system resources, a **profile** is a named set of resource limits and password parameters that restrict database usage and **instance** resources for a user. Profiles can limit the number of concurrent **sessions** for a user, CPU processing time available for each session, and amount of logical I/O available (see "[Buffer I/O](#)" on page 14-10). For example, the `clerk` profile could limit a user to system resources required for clerical tasks.

Note: It is preferable to use Database Resource Manager to limit resources and to use profiles to manage passwords.

Profiles provide a single point of reference for users that share a set of attributes. You can assign a profile to one set of users, and a default profile to all others. Each user has at most one profile assigned at any point in time.

See Also:

- *Oracle Database Security Guide* to learn how to manage resources with profiles
- *Oracle Database SQL Language Reference* for `CREATE PROFILE` syntax and semantics

Authentication

Authentication is the process by which a user presents credentials to the database, which verifies the credentials and allows access to the database. Validating the identity establishes a trust relationship for further interactions. Authentication also enables accountability by making it possible to link access and actions to specific identities.

Oracle Database provides different authentication methods, including the following:

- Authentication by the database

Oracle database can authenticate users using a password, Kerberos ticket, or PKI certificate. Oracle also supports RADIUS-compliant devices for other forms of authentication, including biometrics. The type of authentication must be specified when a user is created in the Oracle database.

- Authentication by the operating system

Some operating systems permit Oracle Database to use information they maintain to authenticate users. After being authenticated by the operating system, users can connect to a database without specifying a user name or password.

Database operations such as shutting down or starting up the database should not be performed by non-administrative database users. These operations require `SYSDBA` or `SYSOPER` privileges (see "[Connection with Administrator Privileges](#)" on page 13-6).

See Also:

- *Oracle Database Security Guide* and *Oracle Database Advanced Security Administrator's Guide* for more information about authentication methods
- *Oracle Database Administrator's Guide* to learn about administrative authentication

Encryption

Encryption is the process of transforming data into an unreadable format using a secret key and an encryption algorithm. Encryption is often used to meet regulatory compliance requirements, such as those associated with the Payment Card Industry Data Security Standard (PCI-DSS) or breach notification laws. For example, credit card numbers, social security numbers, or patient health information must be encrypted.

Network Encryption

Network encryption refers to encrypting data as it travels across the network between a client and server. An intruder can use a network packet sniffer to capture information as it travels on the network, and then spool it to a file for malicious use. Encrypting data on the network prevents this sort of activity.

Transparent Data Encryption

Oracle Advanced Security **transparent data encryption** enables you to encrypt 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.

Transparent data encryption includes industry-standard encryption algorithms such as the Advanced Encryption Standard (AES) and built-in key management.

See Also: *Oracle Database 2 Day + Security Guide* and *Oracle Database Advanced Security Administrator's Guide*

Access Control

Oracle Database provides many techniques to control access to data. This section summarizes some of these techniques.

Oracle Database Vault

Oracle Database Vault is a security option that restricts privileged user access to application data. You can use Oracle Database Vault to control when, where, and how the databases, data, and applications are accessed. Thus, you can address common security problems such as protecting against insider threats, complying with regulatory requirements, and enforcing separation of duty.

See Also: *Oracle Database 2 Day + Security Guide* and *Oracle Database Vault Administrator's Guide*

Virtual Private Database (VPD)

Virtual Private Database (VPD) enables you to enforce security at the row and column level. A **security policy** establishes methods for protecting a database from accidental or malicious destruction of data or damage to the database infrastructure.

VPD is useful when security protections such as privileges and roles are not sufficiently fine-grained. For example, you can allow all users to access the `employees` table, but create security policies to restrict access to employees in the same department as the user.

Essentially, the database adds a dynamic `WHERE` clause to a SQL statement issued against the table, **view**, or **synonym** to which an Oracle VPD security policy was applied. The `WHERE` clause allows only users whose credentials pass the security policy to access the protected data.

See Also: *Oracle Database 2 Day + Security Guide* and *Oracle Database Security Guide*

Oracle Label Security (OLS)

Oracle Label Security (OLS) is a security option that enables you to assign data classification and control access using **security labels**. You can assign a label to both data and users.

When assigned to data, the label can be attached as a hidden column to existing tables, providing transparency to existing SQL. For example, rows that contain highly sensitive data can be labeled `HIGHLY SENSITIVE`, while rows that are less sensitive can be labeled `SENSITIVE`, and so on. When a user attempts to access data, OLS compares the user label with the data label and determines whether access should be granted. Unlike VPD, OLS provides an out-of-the-box security policy and the metadata repository for defining and storing labels.

See Also: *Oracle Database 2 Day + Security Guide* and *Oracle Label Security Administrator's Guide*

Monitoring

Oracle Database provides multiple tools and techniques for monitoring user activity.

Database Auditing

Database auditing is the monitoring and recording of selected user database actions. You can use **standard auditing** to audit SQL statements, privileges, schemas, objects, and network and multitier activity. Alternatively, you can use **fine-grained auditing** to monitor specific database activities, such as actions on a database table or times that activities occur. For example, you can audit a table accessed after 9:00 p.m.

Reasons for using auditing include:

- Enabling future accountability for current actions
- Deterring users (or others, such as intruders) from inappropriate actions based on their accountability
- Investigating, monitoring, and recording suspicious activity
- Addressing auditing requirements for compliance

See Also:

- *Oracle Database 2 Day + Security Guide* and *Oracle Database Security Guide* to learn how to enable and disable auditing
- *Oracle Label Security Administrator's Guide* to learn about Oracle Label Security auditing, which supplements standard auditing

Oracle Audit Vault

Oracle Audit Vault enables you to consolidate, report, and configure alerts for audited data. You can consolidate audit data generated by Oracle Database and other relational databases. You can also use Oracle Audit Vault to monitor audit settings on target databases.

See Also: *Oracle Audit Vault Administrator's Guide*

Enterprise Manager Auditing Support

Oracle Enterprise Manager (Enterprise Manager) enables you to view and configure audit-related initialization parameters. Also, you can administer objects when auditing statements and schema objects. For example, Enterprise Manager enables you to display and search for the properties of current audited statements, privileges, and objects. You can enable and disable auditing as needed.

Overview of High Availability

Availability is the degree to which an application, service, or functionality is available on demand. For example, an **OLTP** database used by an online bookseller is available to the extent that it is accessible by customers making purchases. Reliability, recoverability, timely error detection, and continuous operations are the primary characteristics of **high availability**.

The importance of high availability in a database environment is tied to the cost of **downtime**, which is the time that a resource is unavailable. Downtime can be categorized as either planned or unplanned. The main challenge when designing a highly available environment is examining all possible causes of downtime and developing a plan to deal with them.

See Also: *Oracle Database High Availability Overview* for an introduction to high availability

High Availability and Unplanned Downtime

Oracle Database provides high availability solutions to prevent, tolerate, and reduce downtime for all types of unplanned failures. Unplanned downtime can be categorized by its causes:

- [Site Failures](#)
- [Computer Failures](#)
- [Storage Failures](#)
- [Data Corruption](#)
- [Human Errors](#)

See Also: *Oracle Database High Availability Overview* to learn about protecting against unplanned downtime

Site Failures

A **site failure** occurs when an event causes all or a significant portion of an application to stop processing or slow to an unusable service level. A site failure may affect all processing at a data center, or a subset of applications supported by a data center. Examples include an extended site-wide power or network failure, a natural disaster making a data center inoperable, or a malicious attack on operations or the site.

The simplest form of protection against site failures is to create database backups using RMAN and store them offsite. You can restore the database to another host. However, this technique can be time-consuming, and the backup may not be current. Maintaining one or more standby databases in a Data Guard environment enables you to provide continuous database service if the production site fails.

See Also:

- *Oracle Database High Availability Overview* to learn about site failures
- *Oracle Database Backup and Recovery User's Guide* for information on RMAN and backup and recovery solutions
- *Oracle Data Guard Concepts and Administration* for an introduction to standby databases

Computer Failures

A **computer failure** outage occurs when the system running the database becomes unavailable because it has shut down or is no longer accessible. Examples of computers failures include hardware and operating system failures.

The following Oracle features protect against or help respond to computer failures:

- Enterprise Grids

In an **Oracle Real Applications Cluster (Oracle RAC)** environment, Oracle Database runs on two or more systems in a cluster while concurrently accessing a single shared database. A single database system spans multiple hardware systems yet appears to the application as a single database. See "[Overview of Grid Computing](#)" on page 17-11.

- Oracle Data Guard

Data Guard enables you to maintain a copy of a production database, called a **standby database**, that can reside on a different continent or in the same data center. If the primary database is unavailable because of an outage, then Data Guard can switch any standby database to the primary role, minimizing downtime. See *Oracle Data Guard Concepts and Administration*.

- Oracle Restart

Components in the Oracle Database software stack, including the database instance, listener, and **Oracle ASM** instance, can restart automatically after a component failure or whenever the database host computer restarts. Oracle Restart ensures that Oracle components are started in the proper order, in accordance with component dependencies. See *Oracle Database Administrator's Guide* to learn how to configure Oracle Restart.

- Fast Start Fault Recovery

A common cause of unplanned downtime is a system fault or crash. The **fast start fault recovery technology** in Oracle Database automatically bounds database **instance recovery** time. See *Oracle Database Performance Tuning Guide* for information on fast start fault recovery.

See Also: *Oracle Database High Availability Best Practices* to learn how to use High Availability for processes and applications that run in a single-instance database

Storage Failures

A **storage failure** outage occurs when the storage holding some or all of the database contents becomes unavailable because it has shut down or is no longer accessible. Examples of storage failures include the failure of a disk drive or storage array.

In addition to Oracle Data Guard, solutions for storage failures include the following:

- Oracle Automatic Storage Management (Oracle ASM)
Oracle ASM is a vertically integrated file system and volume manager in the database kernel (see "[Oracle Automatic Storage Management \(Oracle ASM\)](#)" on page 11-3). Oracle ASM eliminates the complexity associated with managing data and disks, and simplifies mirroring and the process of adding and removing disks.
- Backup and recovery
The Recovery Manager (RMAN) utility can back up data, restore data from a previous backup, and recover changes to that data up to the time before the failure occurred (see "[Backup and Recovery](#)" on page 18-9).

See Also:

- *Oracle Database 2 Day DBA* to learn how to administer Oracle ASM disks with Oracle Enterprise Manager (Enterprise Manager)
- *Oracle Automatic Storage Management Administrator's Guide* to learn more about Oracle ASM

Data Corruption

A **data corruption** occurs when a hardware, software or network component causes corrupt data to be read or written. For example, a volume manager error causes bad disk read or writes. Data corruptions are rare but can have a catastrophic effect on a database, and therefore a business.

In addition to Data Guard and Recovery Manager, Oracle Database supports the following forms of protection against data corruption:

- Lost write protection
A **data block** lost write occurs when an I/O subsystem acknowledges the completion of the block write when the write did not occur. You can configure the database so that it records buffer cache block reads in the redo log. Lost write detection is most effective when used with Data Guard.
- Data block corruption detection
A **block corruption** is a data block that is not in a recognized Oracle format, or whose contents are not internally consistent. Several database components and utilities, including RMAN, can detect a corrupt block and record it in `V$DATABASE_BLOCK_CORRUPTION`. If the environment uses a real-time standby database, then RMAN can automatically repair corrupt blocks.
- Data Recovery Advisor
Data Recovery Advisor is an Oracle tool that automatically diagnoses data failures, determines and presents appropriate repair options, and executes repairs at the user's request.

See Also:

- *Oracle Database High Availability Best Practices* to learn how to protect against data corruptions
- *Oracle Database Backup and Recovery User's Guide* for information on RMAN and backup and recovery solutions

Human Errors

A **human error** outage occurs when unintentional or malicious actions are committed that cause data in the database to become logically corrupt or unusable. The service level impact of a human error outage can vary significantly depending on the amount and critical nature of the affected data.

Much research cites human error as the largest cause of downtime. Oracle Database provides powerful tools to help administrators quickly diagnose and recover from these errors. It also includes features that enable end users to recover from problems without administrator involvement.

Oracle Database recommends the following forms of protection against human error:

- **Restriction of user access**
The best way to prevent errors is to restrict user access to data and services. Oracle Database provides a wide range of security tools to control user access to application data by authenticating users and then allowing administrators to grant users only those privileges required to perform their duties (see "[Overview of Database Security](#)" on page 17-1).
- **Oracle Flashback Technology**
Oracle Flashback Technology is a family of human error correction features in Oracle Database. Oracle Flashback provides a SQL interface to quickly analyze and repair human errors. For example, you can perform:
 - Fine-grained surgical analysis and repair for localized damage
 - Rapid correction of more widespread damage
 - Recovery at the row, transaction, table, tablespace, and database level
- **Oracle LogMiner**
Oracle LogMiner is a relational tool that enables [online redo log files](#) to be read, analyzed, and interpreted using SQL (see "[Oracle LogMiner](#)" on page 18-8).

See Also:

- *Oracle Database High Availability Best Practices* to learn how to recover from human errors
- *Oracle Database Backup and Recovery User's Guide* and *Oracle Database Advanced Application Developer's Guide* to learn more about Oracle Flashback features
- *Oracle Database Utilities* to learn more about Oracle LogMiner

High Availability and Planned Downtime

Planned downtime can be just as disruptive to operations, especially in global enterprises that support users in multiple time zones. In this case, it is important to design a system to minimize planned interruptions such as routine operations, periodic maintenance, and new deployments.

Planned downtime can be categorized by its causes:

- [System and Database Changes](#)
- [Data Changes](#)
- [Application Changes](#)

See Also: *Oracle Database High Availability Overview* to learn about features and solutions for planned downtime

System and Database Changes

Planned system changes occur when you perform routine and periodic maintenance operations and new deployments, including scheduled changes to the operating environment that occur outside of the organizational data structure in the database. Examples include adding or removing CPUs and cluster nodes (a **node** is a computer on which a database instance resides), upgrading system hardware or software, and migrating the system platform.

Oracle Database provides **dynamic resource provisioning** as a solution to planned system and database changes:

- **Dynamic reconfiguration of the database**

Oracle Database dynamically accommodates various changes to hardware and database configurations, including adding and removing processors from an SMP server and adding and remove storage arrays using Oracle ASM. For example, Oracle Database monitors the operating system to detect changes in the number of CPUs. If the `CPU_COUNT` initialization parameter is set to the default, then the database workload can dynamically take advantage of newly added processors.
- **Autotuning memory management**

Oracle Database uses a noncentralized policy to free and acquire memory in each subcomponent of the **SGA** and the **PGA**. Oracle Database autotunes memory by prompting the operating system to transfer granules of memory to components that require it. See "[Memory Management](#)" on page 18-15.
- **Automated distributions of data files, control files, and online redo log files**

Oracle ASM automates and simplifies the layout of data files, control files, and log files by automatically distributing them across all available disks. See *Oracle Automatic Storage Management Administrator's Guide* to learn more about Oracle ASM.

Data Changes

Planned data changes occur when there are changes to the logical structure or physical organization of Oracle Database objects. The primary objective of these changes is to improve performance or manageability. Examples include table redefinition, adding table **partitions**, and creating or rebuilding indexes.

Oracle Database minimizes downtime for data changes through online reorganization and redefinition. This architecture enables you to perform the following tasks when the database is open:

- Perform **online table redefinition**, which enables you to make table structure modifications without significantly affecting the availability of the table
- Create, analyze, and reorganize **indexes** (see [Chapter 3, "Indexes and Index-Organized Tables"](#))
- Move table partitions (see "[Overview of Partitions](#)" on page 4-1)

See Also: *Oracle Database Administrator's Guide* to learn how to change data structures online

Application Changes

Planned application changes may include changes to data, schemas, and programs. The primary objective of these changes is to improve performance, manageability, and functionality. An example is an application upgrade.

Oracle Database supports the following solutions for minimizing application downtime required to make changes to an application's database objects:

- Rolling patch updates

Oracle Database supports the application of patches to the nodes of an Oracle RAC system in a rolling fashion. See *Oracle Database High Availability Best Practices*.

- Rolling release upgrades

Oracle Database supports the installation of database software upgrades, and the application of patchsets, in a rolling fashion—with near zero database downtime—by using Data Guard SQL Apply and logical standby databases. See *Oracle Database Upgrade Guide*.

- Edition-based redefinition

Edition-based redefinition enables you to upgrade the database objects of an application while the application is in use, thus minimizing or eliminating down time. Oracle Database accomplishes this task by changing (redefining) database objects in a private environment known as an **edition**. See *Oracle Database Advanced Application Developer's Guide*.

- **DDL** with the default `WAIT` option

DDL commands require exclusive **locks** on internal structures (see "DDL Locks" on page 9-24). In previous releases, DDL commands would fail if they could not obtain the locks. DDL specified with the `WAIT` option resolves this issue. See *Oracle Database High Availability Overview*.

- Creation of triggers in a disabled state

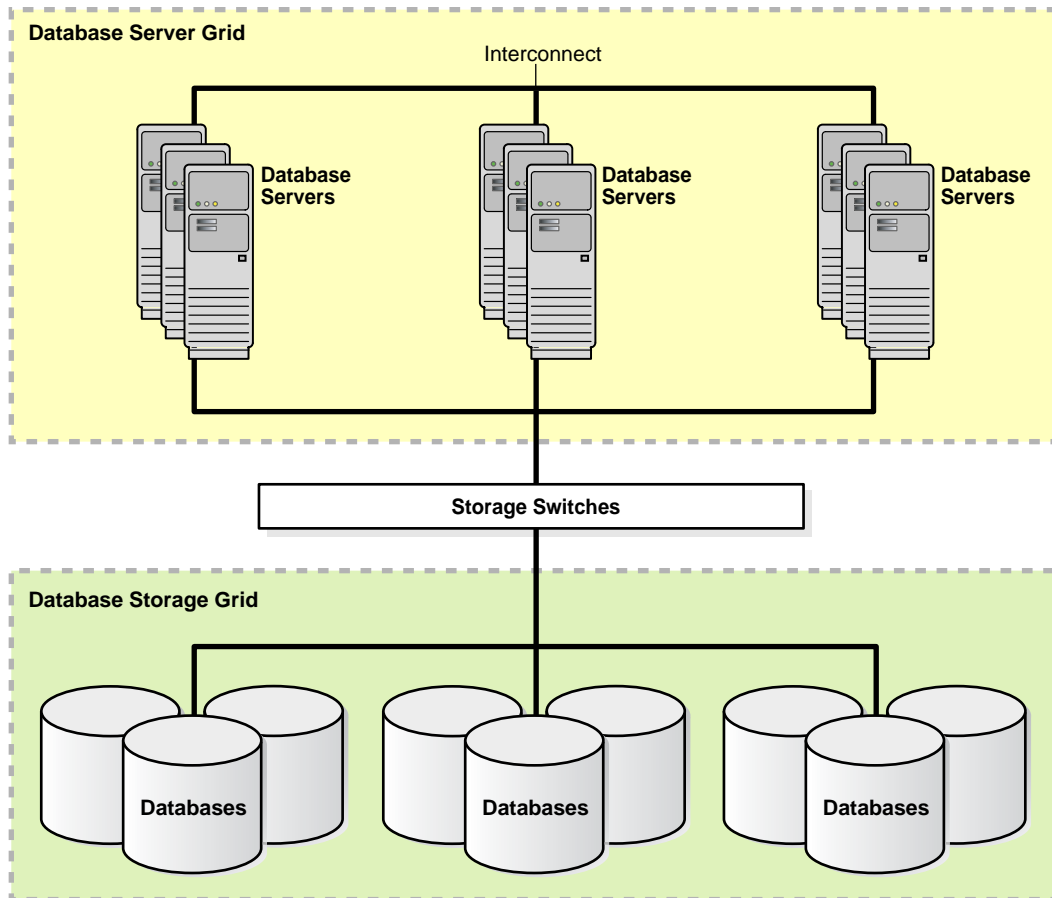
You can create a **trigger** in the disabled state so that you can ensure that your code compiles successfully before you enable the trigger. See *Oracle Database PL/SQL Language Reference*.

Overview of Grid Computing

Grid computing is a computing architecture that effectively pools large numbers of servers and storage into a flexible, on-demand resource for all enterprise computing needs. A **Database Server Grid** is a collection of commodity servers connected together to run on one or more databases. A **Database Storage Grid** is a collection of low-cost modular storage arrays combined together and accessed by the computers in the Database Server Grid.

With the Database Server and Storage Grid, you can build a pool of system resources. You can dynamically allocate and deallocate these resources based on business priorities.

[Figure 17-2](#) illustrates the Database Server Grid and Database Storage Grid in a Grid enterprise computing environment.

Figure 17–2 Grid Computing Environment**See Also:**

- *Oracle Database High Availability Overview* for an overview of Grid Computing
- <http://www.gridforum.org/> to learn about the standards organization Global Grid Forum (GGF)

Database Server Grid

Oracle Real Application Clusters (Oracle RAC) enables multiple instances that are linked by an interconnect to share access to an Oracle database. In an Oracle RAC environment, Oracle Database runs on two or more systems in a cluster while concurrently accessing a single shared database. Oracle RAC enables a Database Server Grid by providing a single database that spans multiple low-cost servers yet appears to the application as a single, unified database system.

Oracle Clusterware is software that enables servers to operate together as if they are one server. Each server looks like any standalone server. However, each server has additional processes that communicate with each other so that separate servers work together as if they were one server. Oracle Clusterware provides all of the features required to run the cluster, including node membership and messaging services.

See Also:

- *Oracle Database 2 Day + Real Application Clusters Guide* for an introduction to Oracle Clusterware and Oracle RAC
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn how to manage an Oracle RAC database
- *Oracle Clusterware Administration and Deployment Guide* to learn how to administer and deploy Oracle Clusterware

Scalability

In a Database Server Grid, Oracle RAC enables you to add nodes to the cluster as the demand for capacity increases. The Cache Fusion technology implemented in Oracle RAC enables you to scale capacity without changing your applications. Thus, you can scale the system incrementally to save costs and eliminate the need to replace smaller single-node systems with larger ones.

You can incrementally add nodes to a cluster instead of replacing existing systems with larger nodes. Grid Plug and Play simplifies addition and removal of nodes from a cluster, making it easier to deploy clusters in a dynamically provisioned environment. Grid Plug and Play also enables databases and services to be managed in a location-independent manner. SCAN enables clients to connect to the database service without regard for its location within the grid.

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about Cache Fusion
- *Oracle Database Installation Guide* to learn how to install Grid Plug and Play

Fault Tolerance

Fault tolerance is the protection provided by a high availability architecture against the failure of a component in the architecture. A key advantage of the Oracle RAC architecture is the inherent fault tolerance provided by multiple nodes. Because the physical nodes run independently, the failure of one or more nodes does not affect other nodes in the cluster.

Failover can happen to any node on the Grid. In the extreme case, an Oracle RAC system provides database service even when all but one node is down. This architecture allows a group of nodes to be transparently put online or taken offline, for maintenance, while the rest of the cluster continues to provide database service.

Oracle RAC provides built-in integration with Oracle Clients and connection pools. With this capability, an application is immediately notified of any failure through the pool that terminates the connection. The application avoids waiting for a TCP timeout and can immediately take the appropriate recovery action. Oracle RAC integrates the **listener** with Oracle Clients and the connection pools to create optimal application throughput. Oracle RAC can balance cluster workload based on the load at the time of the transaction.

See Also:

- ["Database Resident Connection Pooling"](#) on page 16-14
- *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about automatic workload management
- *Oracle Database High Availability Best Practices* for an overview of fault tolerance in Oracle RAC

Services

Oracle RAC supports **services** that can group database workloads and route work to the optimal instances assigned to offer the services. A service represents the workload of applications with common attributes, performance thresholds, and priorities.

You define and apply business policies to these services to perform tasks such as to allocate nodes for times of peak processing or to automatically handle a server failure. Using services ensures the application of system resources where and when they are needed to achieve business goals.

Services are integrated with the Database Resource Manager, which enables you to restrict the resources that are used by a service within an instance. In addition, Oracle Scheduler jobs can run using a service, as opposed to using a specific instance.

See Also:

- *Oracle Database 2 Day + Real Application Clusters Guide* to learn about Oracle services
- *Oracle Database Administrator's Guide* to learn about the Database Resource Manager and Oracle Scheduler

Database Storage Grid

A DBA or storage administrator can use the Oracle ASM interface to specify the disks within the Database Storage Grid that ASM should manage across all server and storage platforms. ASM partitions the disk space and evenly distributes the data across the disks provided to ASM. Additionally, ASM automatically redistributes data as disks from storage arrays are added or removed from the Database Storage Grid.

See Also:

- ["Oracle Automatic Storage Management \(Oracle ASM\)"](#) on page 11-3
- *Oracle Database High Availability Overview* for an overview of the Database Storage Grid
- *Oracle Automatic Storage Management Administrator's Guide* for more information about clustered Oracle ASM

Overview of Data Warehousing and Business Intelligence

A **data warehouse** is a relational database designed for query and analysis rather than for **transaction** processing. For example, a data warehouse could track historical stock prices or income tax records. A warehouse usually contains data derived from historical transaction data, but it can include data from other sources.

A data warehouse environment includes several tools in addition to a relational database. A typical environment includes an **ETL** solution, an **OLAP** engine, Oracle

Warehouse Builder, client analysis tools, and other applications that gather data and deliver it to users.

Data Warehousing and OLTP

A common way of introducing data warehousing is to refer to the characteristics of a data warehouse as set forth by William Inmon¹:

- **Subject-Oriented**
Data warehouses enable you to define a database by subject matter, such as sales.
- **Integrated**
Data warehouses must put data from disparate sources into a consistent format. They must resolve such problems as naming conflicts and inconsistencies among units of measure. When they achieve this goal, they are said to be integrated.
- **Nonvolatile**
The purpose of a warehouse is to enable you to analyze what has occurred. Thus, after data has entered into the warehouse, data should not change.
- **Time-Variant**
The focus of a data warehouse is on change over time.

Data warehouses and OLTP database have different requirements. For example, to discover trends in business, data warehouses must maintain large amounts of data. In contrast, good performance requires historical data to be moved regularly from OLTP systems to an archive. [Table 17-1](#) lists differences between data warehouses and OLTP.

Table 17-1 Data Warehouses and OLTP Systems

Characteristics	Data Warehouse	OLTP
Workload	Designed to accommodate ad hoc queries. You may not know the workload of your data warehouse in advance, so it should be optimized to perform well for a wide variety of possible queries.	Supports only predefined operations. Your applications might be specifically tuned or designed to support only these operations.
Data modifications	Updated on a regular basis by the ETL process using bulk data modification techniques. End users of a data warehouse do not directly update the database.	Subject to individual DML statements routinely issued by end users. The OLTP database is always up to date and reflects the current state of each business transaction.
Schema design	Uses denormalized or partially denormalized schemas (such as a star schema) to optimize query performance.	Uses fully normalized schemas to optimize DML performance and to guarantee data consistency.
Typical operations	A typical query scans thousands or millions of rows. For example, a user may request the total sales for all customers last month.	A typical operation accesses only a handful of records. For example, a user may retrieve the current order for a single customer.
Historical data	Stores many months or years of data to support historical analysis.	Stores data from only a few weeks or months. Historical data retained as needed to meet the requirements of the current transaction.

¹ *Building the Data Warehouse*, John Wiley and Sons, 1996.

See Also:

- *Oracle Database Data Warehousing Guide* for a more detailed description of a database warehouse
- *Oracle Database VLDB and Partitioning Guide* for a more detailed description of an OLTP system

Data Warehouse Architecture

Data warehouses and their architectures vary depending on the business requirements. This section describes common data warehouse architectures.

Data Warehouse Architecture (Basic)

Figure 17-3 shows a simple architecture for a data warehouse. End users directly access data that was transported from several source systems to the data warehouse.

Figure 17-3 Architecture of a Data Warehouse

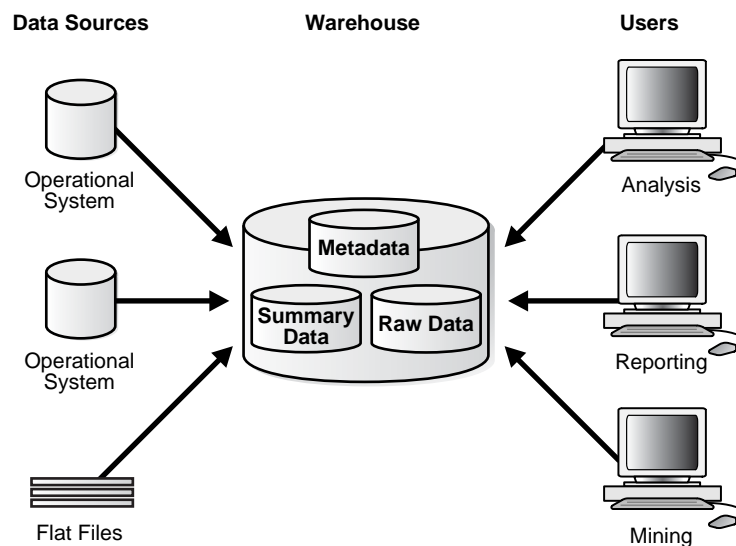


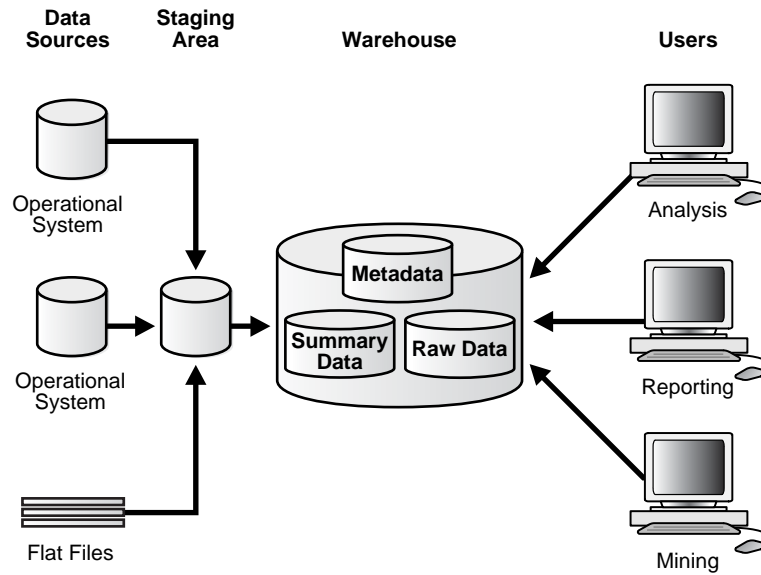
Figure 17-3 shows both the metadata and raw data of a traditional OLTP system and summary data. A **summary** is an aggregate view that improves query performance by precalculating expensive **joins** and aggregation operations and storing the results in a table. For example, a summary table can contain the sums of sales by region and by product. Summaries are also called **materialized views**.

See Also: *Oracle Database Data Warehousing Guide* to learn about basic materialized views

Data Warehouse Architecture (with a Staging Area)

In the architecture shown in Figure 17-3, operational data must be cleaned and processed before being put into the warehouse. Figure 17-4 shows a data warehouse with a **staging area**, which is a place where data is preprocessed before entering the warehouse. A staging area simplifies the tasks of building summaries and managing the warehouse.

Figure 17-4 Architecture of a Data Warehouse with a Staging Area



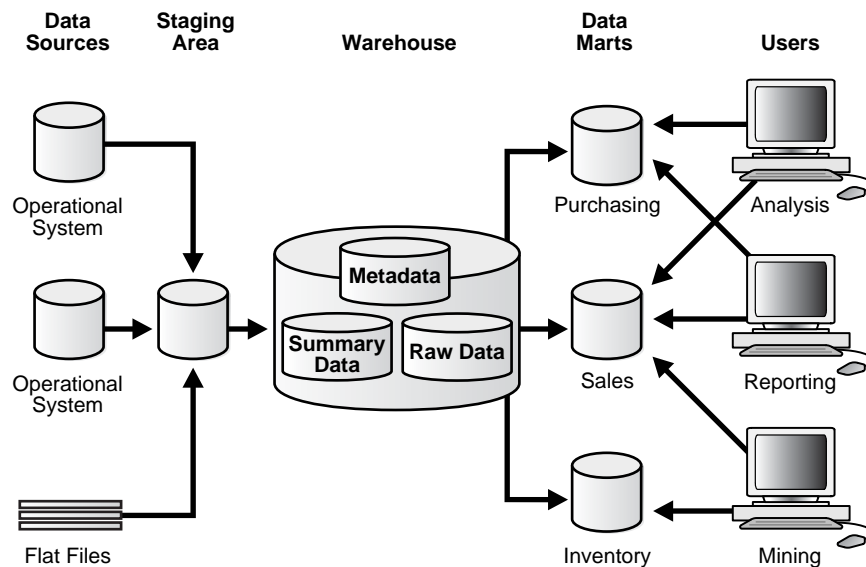
See Also: *Oracle Database Data Warehousing Guide* to learn about different transportation mechanisms

Data Warehouse Architecture (with a Staging Area and Data Marts)

You may want to customize your warehouse architecture for different groups within your organization. You can achieve this goal by transporting data in the warehouse to **data marts**, which are independent databases designed for a specific business or project. Typically, data marts include many summary tables.

Figure 17-5 separates purchasing, sales, and inventory information into independent data marts. A financial analyst can query the data marts for historical information about purchases and sales.

Figure 17-5 Architecture of a Data Warehouse with a Staging Area and Data Marts



See Also: *Oracle Database Data Warehousing Guide* to learn about transformation mechanisms

Overview of Extraction, Transformation, and Loading (ETL)

The process of extracting data from source systems and bringing it into the warehouse is commonly called ETL: extraction, transformation, and loading. ETL refers to a broad process rather than three well-defined steps.

In a typical scenario, data from one or more operational systems is extracted and then physically transported to the target system or an intermediate system for processing. Depending on the method of transportation, some transformations can occur during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the `SELECT` statement.

Oracle Database is not itself an ETL tool. However, Oracle Database provides a rich set of capabilities usable by ETL tools such as Oracle Warehouse Builder and customized ETL solutions. ETL capabilities provided by Oracle Database include:

- **Transportable tablespaces**

You can transport **tablespaces** between different computer architectures and operating systems. Transportable tablespaces are the fastest way for moving large volumes of data between two Oracle databases. See *Oracle Database Administrator's Guide* to learn about transportable tablespaces.

- **Table functions**

A **table function** can produce a set of rows as output and can accept a set of rows as input. Table functions provide support for pipelined and parallel execution of transformations implemented in PL/SQL, C, or Java without requiring the use of intermediate staging tables. See *Oracle Database Data Warehousing Guide* to learn about table functions.

- **External tables**

External tables enable external data to be joined directly and in parallel without requiring it to be first loaded in the database (see "**External Tables**" on page 2-16). Thus, external tables enable the pipelining of the loading phase with the transformation phase.

- **Table compression**

To reduce disk use and memory use, you can store tables and partitioned tables in a compressed format (see "**Table Compression**" on page 2-19). The use of **table compression** often leads to a better scaleup for read-only operations and faster query execution.

- **Change Data Capture**

This feature efficiently identifies and captures data that has been added to, updated in, or removed from, relational tables and makes this change data available for use by applications or individuals.

See Also:

- *Oracle Database Data Warehousing Guide* to learn about Change Data Capture
- *Oracle Warehouse Builder Data Modeling, ETL, and Data Quality Guide* for an overview of ETL

Business Intelligence

Business intelligence is the analysis of an organization's information as an aid to making business decisions. Business intelligence and analytical applications are dominated by actions such as drilling up and down hierarchies and comparing aggregate values. Oracle Database provides several technologies to support business intelligence operations.

Analytic SQL

Oracle Database has introduced many SQL operations for performing analytic operations. These operations include ranking, moving averages, cumulative sums, ratio-to-reports, and period-over-period comparisons. For example, Oracle Database supports the following forms of analytic SQL:

- SQL for aggregation

Aggregate functions such as `COUNT` return a single result row based on groups of rows rather than on single rows. Aggregation is fundamental to data warehousing. To improve aggregation performance in a warehouse, the database provides extensions to the `GROUP BY` clause to make querying and reporting easier and faster. See *Oracle Database Data Warehousing Guide* to learn about aggregation.
- SQL for analysis

Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. Oracle has advanced SQL analytical processing capabilities using a family of analytic SQL functions. For example, these analytic functions enable you to calculate rankings and percentiles and moving windows. See *Oracle Database Data Warehousing Guide* to learn about SQL for analysis and reporting.
- SQL for modeling

With the `MODEL` clause, you can create a multidimensional array from query results and apply rules to this array to calculate new values. For example, you can partition data in a sales view by country and perform a model computation, as defined by multiple rules, on each country. One rule could calculate the sales of a product in 2008 as the sum of sales in 2006 and 2007. See *Oracle Database Data Warehousing Guide* to learn about SQL modeling.

See Also: *Oracle Database SQL Language Reference* to learn about SQL functions

OLAP

Oracle **online analytical processing (OLAP)** provides native multidimensional storage and rapid response times when analyzing data across multiple **dimensions**. OLAP enables analysts to quickly obtain answers to complex, iterative queries during interactive sessions.

Oracle OLAP has the following primary characteristics:

- Oracle OLAP is integrated in the database so that you can use standard SQL administrative, querying, and reporting tools.
- The OLAP engine runs within the kernel of Oracle Database.
- Dimensional objects are stored in Oracle Database in their native multidimensional format.

- Cubes and other dimensional objects are first class data objects represented in the Oracle [data dictionary](#).
- Data security is administered in the standard way, by granting and revoking privileges to Oracle Database users and roles.

Oracle OLAP offers the power of simplicity: one database, standard administration and security, and standard interfaces and development tools.

See Also:

- ["Overview of Dimensions"](#) on page 4-21
- *Oracle OLAP User's Guide* for an overview of Oracle OLAP

Data Mining

Data mining involves automatically searching large stores of data for patterns and trends that go beyond simple analysis. Data mining uses sophisticated mathematical algorithms to segment data and evaluate the probability of future events. Typical applications of data mining include call centers, ATMs, E-business relational management (ERM), and business planning.

With Oracle Data Mining, the data, data preparation, model building, and model scoring results all remain in the database. Oracle Data Mining supports a PL/SQL API, a Java API, SQL functions for model scoring, and a GUI called Oracle Data Miner. Thus, Oracle Database provides an infrastructure for application developers to integrate data mining seamlessly with database applications.

See Also: *Oracle Data Mining Concepts*

Overview of Oracle Information Integration

As an organization evolves, it becomes increasingly important for it to be able to share information among multiple databases and applications. The basic approaches to sharing information are as follows:

- Consolidation
You can consolidate the information into a single database, which eliminates the need for further integration. Oracle RAC, Grid computing, and Oracle VPD can enable you to consolidate information into a single database.
- Federation
You can leave information distributed, and provide tools to **federate** this information, making it appear to be in a single virtual database.
- Sharing
You can share information, which lets you maintain the information in multiple data stores and applications.

This section focuses on Oracle solutions for federating and sharing information.

See Also: *Oracle Database 2 Day + Data Replication and Integration Guide* for an introduction to data replication and integration

Federated Access

The foundation of federated access is a **distributed environment**, which is a network of disparate systems that seamlessly communicate with each other. Each system in the

environment is called a **node**. The system to which a user is directly connected is called the **local system**. Additional systems accessed by this user are **remote systems**.

A distributed environment enables applications to access and exchange data from the local and remote systems. All the data can be simultaneously accessed and modified.

Distributed SQL

Distributed SQL synchronously accesses and updates data distributed among multiple databases. An Oracle distributed database system can be transparent to users, making it appear as a single Oracle database.

Distributed SQL includes distributed queries and **distributed transactions**. The Oracle distributed database architecture provides query and transaction transparency. For example, standard DML statements work just as they do in a non-distributed database environment. Additionally, applications control transactions using the standard SQL statements COMMIT, SAVEPOINT, and ROLLBACK.

See Also:

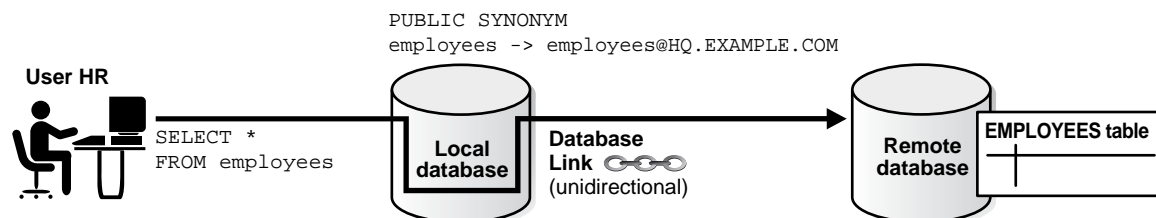
- ["Overview of Distributed Transactions"](#) on page 10-12
- *Oracle Database 2 Day + Data Replication and Integration Guide* to learn about distributed SQL
- *Oracle Database Administrator's Guide* to learn how to manage distributed transactions

Database Links

A **database link** is a connection between two physical databases that enables a client to access them as one logical database. Oracle Database uses database links to enable users on one database to access objects in a remote database. A local user can access a link to a remote database without being a user on the remote database.

[Figure 17-6](#) shows an example of user hr accessing the employees table on the remote database with the global name hr.example.com. The employees synonym hides the identity and location of the remote schema object.

Figure 17-6 Database Link



See Also: *Oracle Database Administrator's Guide* to learn about database links

Information Sharing

At the heart of any integration is the sharing of data among applications in the enterprise. **Oracle Streams** is the asynchronous information sharing infrastructure in Oracle Database. This infrastructure enables the propagation and management of data, transactions, and events in a data stream either within a database, or from one database to another.

Oracle Streams includes replication and messaging. **Replication** is the process of sharing database objects and data at multiple databases. **Messaging** is the sharing of information between applications and users.

See Also:

- *Oracle Streams Concepts and Administration*
- *Oracle Streams Replication Administrator's Guide*

Oracle Streams Replication

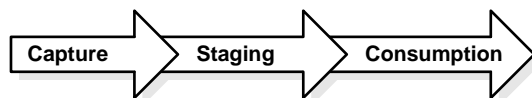
In **Oracle Streams replication**, a change to a database object at one database can be shared with other databases in the replication environment. For example, Oracle Streams propagates an update to an `employees` table to an identical `employees` table in a different database. In this way, the database objects and data are kept synchronized at all databases in the replication environment.

Typical uses for Oracle Streams replication include:

- Creating a reporting site to offload processing from a primary OLTP site.
- Providing load balancing and improved scalability and availability for a call center or similar application.
- Providing site autonomy between locations to satisfy certain common business requirements.
- Transforming and consolidating data from multiple locations.
- Replicating data between different platforms and Oracle Database releases, and across a wide area network (WAN).

Oracle Streams Information Flow The architecture of Oracle Streams is very flexible. [Figure 17-7](#) depicts the basic information flow in a replication environment.

Figure 17-7 Oracle Streams Information Flow



As shown in [Figure 17-7](#), Oracle Streams contains the following basic elements:

- **Capture**
Oracle Streams can implicitly capture DML and DDL changes. Rules determine which changes are captured. Changes are formatted into **logical change records (LCRs)**, which are messages with a specific format describing a database change.
- **Staging**
LCRs are placed in a staging area, which is a queue that stores and manages captured messages. Message staging provides a holding area with security, as well as auditing and tracking of message data. Propagations can send messages from one queue to another. The queues can reside in the same or different databases.
- **Consumption**
LCRs remain in a staging area until subscribers consume them implicitly or explicitly. An apply process implicitly applies changes encapsulated in LCRs.

Note: Oracle Streams is fully inter-operational with **materialized views**, which you can use to maintain updatable or read-only copies of data (see "Overview of Materialized Views" on page 4-16).

See Also: *Oracle Database 2 Day + Data Replication and Integration Guide* to learn how to replicate data using Oracle Streams

Oracle Streams Replication Environments Oracle Streams enables you to configure many different types of custom replication environments. However, the following types of replication environments are the most common:

- Two-Database

Only two databases share the replicated database objects. The changes made to replicated database objects at one database are captured and sent directly to the other database, where they are applied.

In a **one-way replication environment**, only one database allows changes to the replicated database objects, with the other database containing read-only replicas of these objects. In a **bi-directional replication environment**, both databases can allow changes to the replicated objects. In this case, both databases capture changes to these database objects and send the changes to the other database, where they are applied.

- Hub-and-Spoke

A central database, or **hub**, communicates with secondary databases, or **spokes**. The spokes do not communicate directly with each other. In a hub-and-spoke replication environment, the spokes might or might not allow changes to the replicated database objects.

- N-Way

Each database communicates directly with every other database in the environment. The changes made to replicated database objects at one database are captured and sent directly to each of the other databases in the environment, where they are applied.

See Also: *Oracle Database 2 Day + Data Replication and Integration Guide* to learn more about common replication environments

Oracle Streams Advanced Queuing (AQ)

Oracle Streams Advanced Queuing (AQ) is a robust and feature-rich message queuing system integrated with Oracle Database. When an organization has different systems that must communicate with each other, a messaging environment can provide a standard, reliable way to transport critical information between these systems.

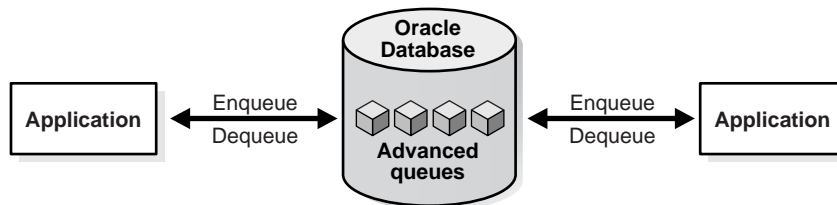
A sample use case is a business that enters orders in an Oracle database at headquarters. When an order is entered, the business uses AQ to send the order ID and order date to a database in a warehouse. These messages alert employees at the warehouse about the orders so that they can fill and ship them.

Message Queuing and Dequeuing Advanced Queuing stores user messages in abstract storage units called **queues**. **Enqueuing** is the process by which **producers** place messages into queues. **Dequeuing** is the process by which **consumers** retrieve messages from queues.

Support for explicit dequeue allows developers to use Oracle Streams to reliably exchange messages. They can also notify applications of changes by leveraging the change capture and propagation features of Oracle Streams.

Figure 17–8 shows a sample application that explicitly enqueues and dequeues messages through Advanced Queuing, enabling it to share information with partners using different messaging systems. After being enqueued, messages can be transformed and propagated before being dequeued to the partner's application.

Figure 17–8 Oracle Streams Message Queuing



Advanced Queuing Features Oracle Streams Advanced Queuing supports all the standard features of message queuing systems. These features include:

- **Asynchronous application integration**
Oracle Streams Advanced Queuing offers several ways to enqueue messages. A capture process or synchronous capture can capture the messages implicitly, or applications and users can capture messages explicitly.
- **Extensible integration architecture**
Many applications are integrated with a distributed hub and spoke model with Oracle Database as the hub. The distributed applications on an Oracle database communicate with queues in the same hub. Multiple applications share the same queue, eliminating the need to add queues to support additional applications.
- **Heterogeneous application integration**
Advanced Queuing provides applications with the full power of the Oracle type system. It includes support for scalar data types, Oracle Database object types with inheritance, `XMLType` with additional **operators** for XML data, and `ANYDATA`.
- **Legacy application integration**
The Oracle Messaging Gateway integrates Oracle Database applications with other message queuing systems, such as Websphere MQ and Tibco.
- **Standards-Based API support**
Oracle Streams Advanced Queuing supports industry-standard APIs: SQL, JMS, and SOAP. Changes made using SQL are captured automatically as messages.

See Also:

- *Oracle Database 2 Day + Data Replication and Integration Guide* to learn how to send messages using Advanced Queuing
- *Oracle Streams Advanced Queuing User's Guide*

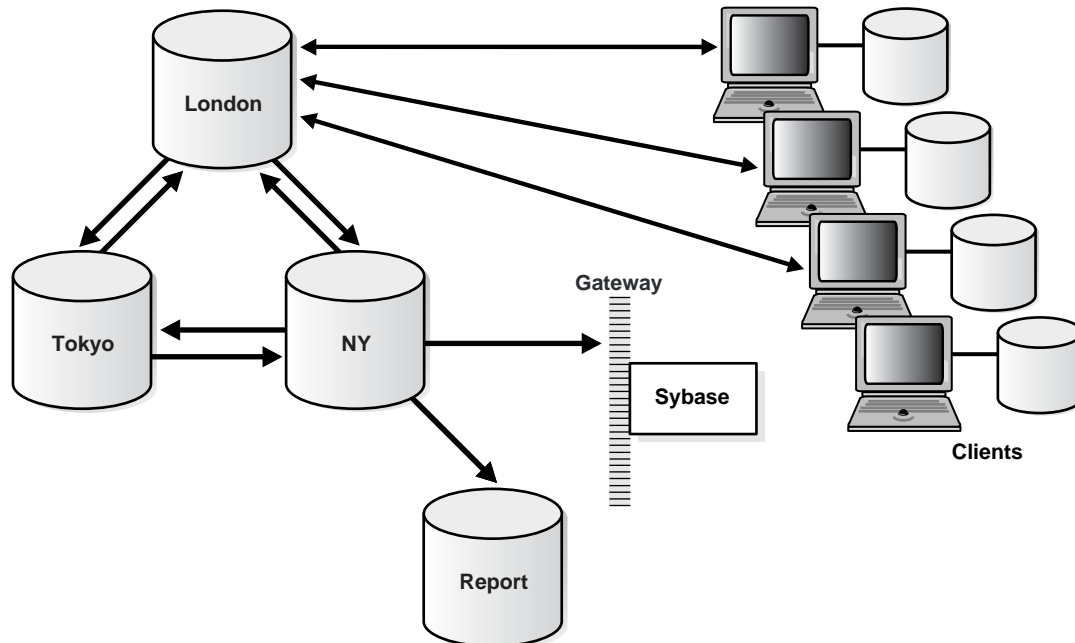
Oracle Streams Use Case

For a sample use case, assume that a company uses Oracle Streams to maintain multiple copies of a corporate Web site. The business requirements include:

- A reporting database must contain the most current data for analysts in a New York office to perform ad hoc querying.
- Updatable materialized views must support the field sales staff.
- Data must be shared with applications hosted on a Sybase database.

Figure 17–9 illustrates this Streams configuration.

Figure 17–9 Streams Configuration



Oracle Streams is used to replicate data in an n-way configuration consisting of sites in New York, London, and Tokyo. At each site, Streams implicit capture collects any changes that occur for subscribed tables in each local region, and stages them locally in the queue. Changes captured in each region are then forwarded to each of the other region's databases. Changes made at each database can be reflected at every other database, providing complete data for the subscribed objects throughout the world.

At each regional database, an Oracle Streams apply process applies the changes automatically. As changes are applied, Oracle Streams checks for and resolves any conflicts. Streams can also be used to exchange data for particular tables with non-Oracle databases. Using the Oracle Database Gateway for Sybase, a Streams apply process applies the changes to a Sybase database using the same mechanisms as it does for Oracle databases.

The reporting database is hosted in New York. This database is a fully functional Oracle database that has a read-only copy of the relevant application tables. The reporting site is not configured to capture changes on these application tables. Oracle Streams imposes no restrictions on the configuration or use of this reporting database.

The London site also serves as the master site for several updatable materialized view sites. Each salesperson receives an updatable copy of the required portion of data. These sites typically only connect once a day to upload their orders and download any changes made after their last refresh.

See Also: *Oracle Database 2 Day + Data Replication and Integration Guide* for examples of configuring Oracle Streams

Concepts for Database Administrators

This chapter contains the following sections:

- [Duties of Database Administrators](#)
- [Tools for Database Administrators](#)
- [Topics for Database Administrators](#)

Duties of Database Administrators

The principal responsibility of a database administrator (DBA) is to make enterprise data available to its users. DBAs must work closely with the developers to ensure that their applications make efficient use of the database, and with system administrators to ensure that physical resources are adequate and used efficiently.

Oracle DBAs are responsible for understanding the Oracle Database architecture and how the database works. DBAs can expect to perform the following tasks:

- Installing, upgrading, and patching Oracle Database software
- Designing databases, including identifying requirements, creating the logical design (conceptual model), and physical database design
- Creating Oracle databases
- Developing and testing a backup and recovery strategy, backing up Oracle databases regularly, and recovering them in case of failures
- Configuring the network environment to enable clients to connect to databases
- Starting up and shutting down the database
- Managing storage for the database
- Managing users and security
- Managing database objects such as tables, indexes, and views
- Monitoring and tuning database performance
- Investigating, gathering diagnostic data for, and reporting to Oracle Support Services any critical database errors
- Evaluating and testing new database features

The preceding tasks, and many others, are described in *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide*.

The types of users and their roles and responsibilities depend on the database environment. A small database may have one DBA. A very large database may divide

the DBA duties among several specialists, for example, security officers, backup operators, and application administrators.

Tools for Database Administrators

Oracle provides several tools for use in administering a database. This section describes some commonly used tools:

- [Oracle Enterprise Manager](#)
- [SQL*Plus](#)
- [Tools for Database Installation and Configuration](#)
- [Tools for Oracle Net Configuration and Administration](#)
- [Tools for Data Movement and Analysis](#)

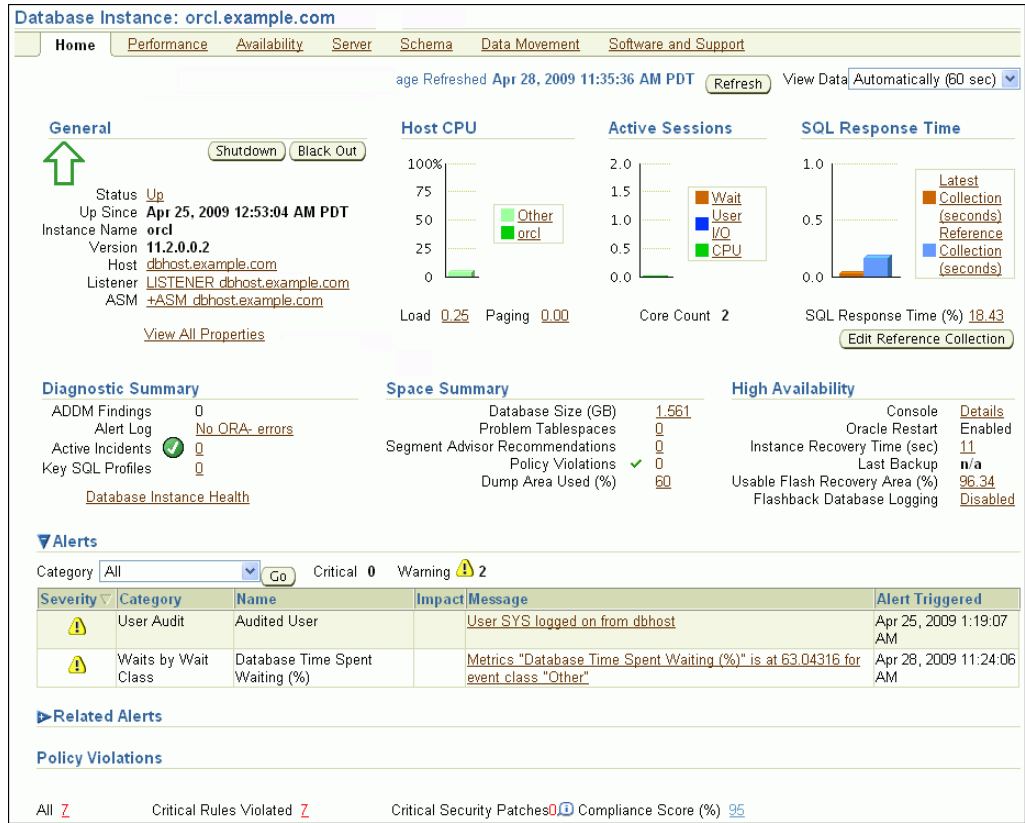
Oracle Enterprise Manager

Oracle Enterprise Manager (Enterprise Manager) is a system management tool that provides centralized management of a database environment. Combining a graphical console, Oracle Management Servers, Oracle Intelligent Agents, common services, and administrative tools, Enterprise Manager provides a comprehensive systems management platform for Oracle products.

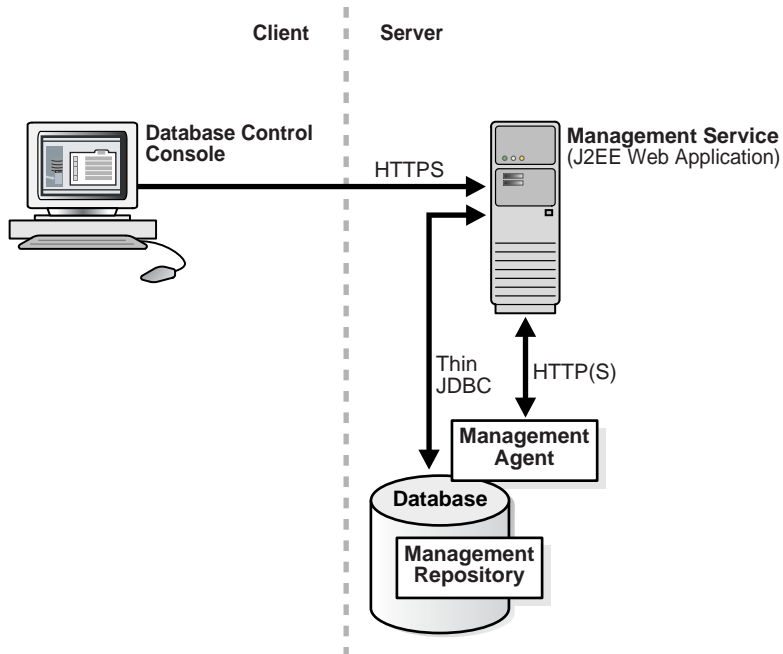
The Web-based **Enterprise Manager Database Control (Database Control)** is the primary tool for managing an Oracle database. It is installed with Oracle Database. You can use Database Control to perform administrative tasks such as:

- Diagnosing, modifying, and tuning the database
- Grouping related targets together to facilitate administration tasks, sharing tasks with other administrators, and scheduling tasks at varying time intervals
- Configuring and managing Oracle Net Services for an **Oracle home** (see "[Overview of Oracle Networking Architecture](#)" on page 16-5)
- Launching integrated Oracle and third-party tools

The following figure shows the Database Home page of Database Control. The subpage links across the top of the page enable you to access performance, availability, and other database administration pages. The subsections of the Database Home page provide information about the environment and status of the database.



The following figure shows the basic architecture of Enterprise Manager. The management repository is stored inside the database. Both the agent and the management service run on the database host. You can run the Database Control Console from any Web browser that can connect securely to the management service.



See Also: *Oracle Database 2 Day DBA* to learn how to administer the database with Enterprise Manager

SQL*Plus

SQL*Plus is an interactive and batch query tool included in every Oracle Database installation. It has a command-line user interface that acts as the client when connecting to the database.

SQL*Plus has its own commands and environment. It enables you to enter and execute SQL, PL/SQL, SQL*Plus and operating system commands to perform tasks such as:

- Formatting, performing calculations on, storing, and printing from query results
- Examining table and object definitions
- Developing and running batch scripts
- Administering a database

You can use SQL*Plus to generate reports interactively, to generate reports as batch processes, and to output the results to text file, to screen, or to HTML file for browsing on the Internet. You can generate reports dynamically using the HTML output facility.

See Also: *Oracle Database 2 Day DBA* and *SQL*Plus User's Guide and Reference* to learn more about SQL*Plus

Tools for Database Installation and Configuration

Oracle provides several tools to simplify the task of installing and configuring Oracle Database software. The tools include:

- Oracle Universal Installer (OUI)
OUI is a GUI utility that enables you to view, install, and deinstall Oracle Database software. Online Help is available to guide you through the installation. See *Oracle Database Installation Guide* to learn how to install Oracle Database software.
- Database Upgrade Assistant (DBUA)
DBUA interactively guides you through a database upgrade and configures the database for the new release. DBUA automates the upgrade by performing all tasks normally performed manually. DBUA makes recommendations for configuration options such as tablespaces and the [online redo log](#). See *Oracle Database 2 Day DBA* to learn how to upgrade a database with DBUA.
- Database Configuration Assistant (DBCA)
DBCA provides a graphical interface and guided workflow for creating and configuring a database. This tool enables you to create a database from Oracle-supplied templates or create your own database and templates. See *Oracle Database Administrator's Guide* to learn how to create a database with DBCA.

Tools for Oracle Net Configuration and Administration

Oracle Net Services provides enterprise wide connectivity solutions in distributed, heterogeneous computing environments. Oracle Net, a component of Oracle Net Services, enables a network session from a client application to an database. You can use the following tools to configure and administer Oracle Net Services:

- Oracle Net Manager
This tool enables you to configure Oracle Net Services for an Oracle home on a local client or server host. You can use Oracle Net Manager to configure naming, naming methods, profiles, and listeners. You can start Oracle Net Manager using the Oracle Enterprise Manager Console or as an independent application.

- Oracle Net Configuration Assistant
This tool runs automatically during software installation. The Assistant enables you to configure basic network components during installation, including listener names and protocol addresses, naming methods, net service names in a `tnsnames.ora` file, and directory server usage.
 - Listener Control Utility
The Listener Control utility enables you to configure listeners to receive client connections (see "[The Oracle Net Listener](#)" on page 16-6). You can access the utility through Enterprise Manager or as a standalone command-line application.
 - Oracle Connection Manager Control Utility
This command-line utility enables you to administer an **Oracle Connection Manager**, which is a router through which a client connection request may be sent either to its next hop or directly to the database. You can use utility commands to perform basic management functions on one or more Oracle Connection Managers. Additionally, you can view and change parameter settings.
- See Also:**
- "[Overview of Oracle Networking Architecture](#)" on page 16-5
 - *Oracle Database Net Services Administrator's Guide* and *Oracle Database Net Services Reference* to learn more about Oracle Net Services tools

Tools for Data Movement and Analysis

Oracle Database includes several utilities to assist in database movement and analysis. For example, you can use database utilities to:

- Load data into Oracle Database tables from operating system files, as explained in "[SQL*Loader](#)" on page 18-5
- Move data and metadata from one database to another database, as explained in "[Oracle Data Pump Export and Import](#)" on page 18-7
- Query redo log files through a SQL interface, as explained in "[Oracle LogMiner](#)" on page 18-8
- Manage Oracle Database diagnostic data, as explained in "[ADR Command Interpreter \(ADRCI\)](#)" on page 18-8

Other tasks include performing physical data structure integrity checks on an offline database or data file with `DBVERIFY`, or changing the database identifier (DBID) or database name for an operational database using the `DBNEWID` utility.

Note: Tools related to backup and recovery are covered in "[Backup and Recovery](#)" on page 18-9.

See Also: *Oracle Database Utilities* to learn about `DBVERIFY` and `DBNEWID`

SQL*Loader

SQL*Loader loads data from external files, called **data files**, into database tables. It has a powerful data parsing engine that puts little limitation on the format of the data in the data file. You can use SQL*Loader to perform tasks such as:

- Loading data from multiple data files into multiple tables

You store the data to be loaded in SQL*Loader data files. The SQL*Loader control file is a text file that contains **DDL** instructions that SQL*Loader uses to determine where to find the data, how to parse and interpret it, where to insert it, and more.

Note: The SQL*Loader data files and control file are unrelated to the Oracle Database **data files** and **control file**.

- Control various aspects of the load operation

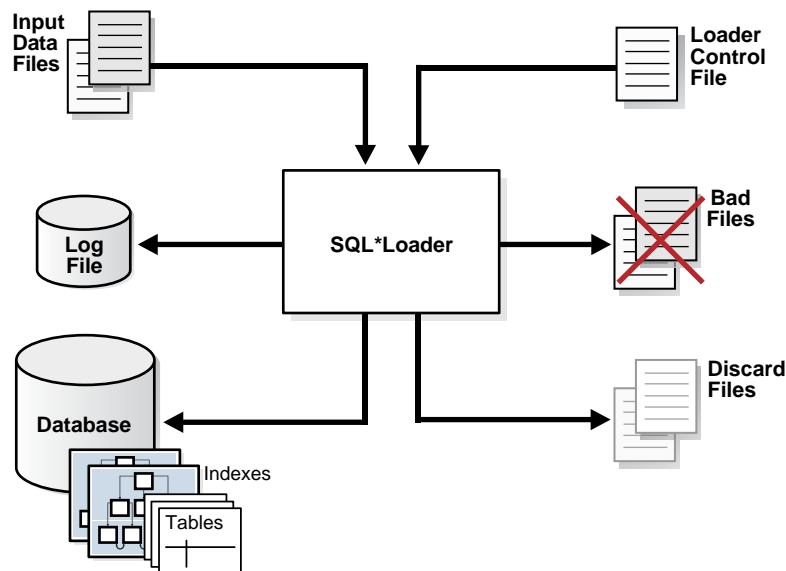
For example, you can selectively load data, specify the data character set (see "Character Sets" on page 19-9), manipulate the data with SQL functions, generate unique sequential key values in specified columns, and so on. You can also generate sophisticated error reports.

- Use either conventional or direct path loading

A **conventional path load** executes SQL INSERT statements to populate tables. In contrast, a **direct path load** eliminates much of the database overhead by formatting data blocks and writing them directly to the database files. Direct writes operate on blocks above the **high water mark** and write directly to disk, bypassing the **database buffer cache**. Direct reads read directly from disk into the **PGA**, again bypassing the buffer cache.

A typical SQL*Loader session takes as input a SQL*Loader control file and one or more data files. The output is an Oracle database, a log file, a bad file, and potentially, a discard file. [Figure 18–1](#) illustrates the flow of a typical SQL*Loader session.

Figure 18–1 SQL*Loader Session



See Also: *Oracle Database 2 Day DBA* and *Oracle Database Utilities* to learn about SQL*Loader

Oracle Data Pump Export and Import

Oracle Data Pump enables high-speed movement of data and metadata from one database to another. This technology is the basis for the following Oracle Database data movement utilities:

- Data Pump Export (Export)

Export is a utility for unloading data and metadata into a set of operating system files called a **dump file set**. The dump file set is made up of one or more binary files that contain table data, database object metadata, and control information.

- Data Pump Import (Import)

Import is a utility for loading an export dump file set into a database. You can also use Import to load a destination database directly from a source database with no intervening files, which allows export and import operations to run concurrently, minimizing total elapsed time.

Oracle Data Pump is made up of the following distinct parts:

- The command-line clients `expdp` and `impdp`

These client make calls to the `DBMS_DATAPUMP` package to perform Oracle Data Pump operations (see "PL/SQL Packages" on page 8-6).

- The `DBMS_DATAPUMP` PL/SQL package, also known as the **Data Pump API**

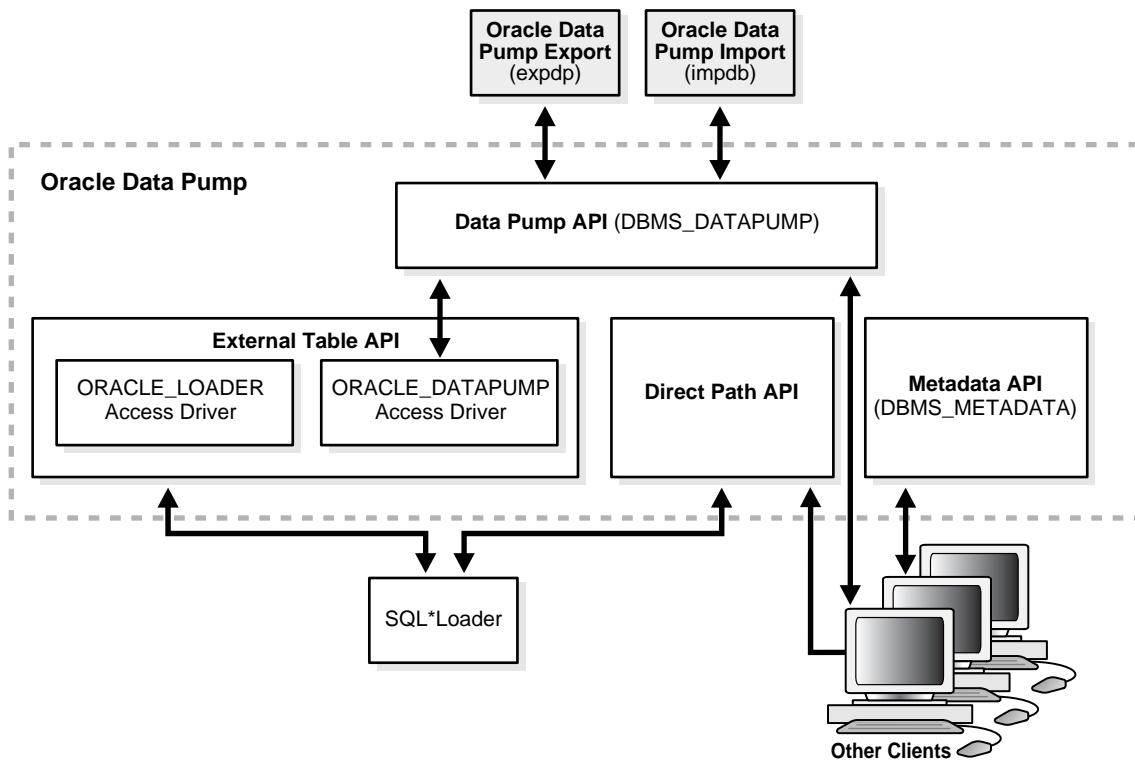
This API provides high-speed import and export functionality.

- The `DBMS_METADATA` PL/SQL package, also known as the **Metadata API**

This API, which stores object definitions in XML, is used by all processes that load and unload metadata.

[Figure 18-2](#) shows how Oracle Data Pump integrates with SQL*Loader and external tables. As shown, SQL*Loader is integrated with the External Table API and the Data Pump API to load data into **external tables** (see "External Tables" on page 2-16). Clients such as Database Control and **transportable tablespaces** can use the Oracle Data Pump infrastructure.

Figure 18–2 Oracle Data Pump Architecture

**See Also:**

- *Oracle Database Utilities* for an overview of Oracle Data Pump
- *Oracle Database PL/SQL Packages and Types Reference* for a description of DBMS_DATAPUMP and DBMS_METADATA

Oracle LogMiner

Oracle LogMiner enables you to query redo log files through a SQL interface. Potential uses for data contained in redo log files include:

- Pinpointing when a logical corruption to a database, such as errors made at the application level, may have begun
- Detecting user error
- Determining what actions you would have to take to perform fine-grained recovery at the **transaction** level
- Using trend analysis to determine which tables get the most updates and inserts
- Analyzing system behavior and auditing database use through the LogMiner comprehensive relational interface to redo log files

LogMiner is accessible through a command-line interface or through the Oracle LogMiner Viewer GUI, which is a part of Enterprise Manager.

See Also: *Oracle Database Utilities* to learn more about LogMiner

ADR Command Interpreter (ADRCI)

ADRCI is a command-line utility that enables you to investigate problems, view health check reports, and package and upload first-failure diagnostic data to Oracle

Support. You can also use the utility to view the names of the trace files in the [Automatic Diagnostic Repository \(ADR\)](#) (ADR) and to view the [alert log](#). ADRCI has a rich command set that you can use interactively or in scripts.

See Also:

- ["Automatic Diagnostic Repository"](#) on page 13-19
- *Oracle Database Utilities* and *Oracle Database Administrator's Guide* for more information on ADR and ADRCI

Topics for Database Administrators

[Chapter 17](#) describes topics important for both developers and DBAs. This section covers topics that are most essential to DBAs and that have not been discussed elsewhere in the manual.

This section contains the following topics:

- [Backup and Recovery](#)
- [Memory Management](#)
- [Resource Management and Task Scheduling](#)
- [Performance Diagnostics and Tuning](#)

Backup and Recovery

Backup and recovery is the set of concepts, procedures, and strategies involved in protecting the database against data loss caused by media failure or users errors. In general, the purpose of a backup and recovery strategy is to protect the database against data loss and reconstruct lost data.

A **backup** is a copy of data. A backup can include crucial parts of the database such as data files, the [server parameter file](#), and control file. A sample backup and recovery scenario is a failed disk drive that causes the loss of a data file. If a backup of the lost file exists, then you can restore and recover it. **Media recovery** refers to the operations involved in restoring data to its state before the loss occurred.

See Also: *Oracle Database 2 Day DBA* and *Oracle Database Backup and Recovery User's Guide* for backup and recovery concepts and tasks

Backup and Recovery Techniques

You can use the following means to back up and recover an Oracle database:

- **Recovery Manager (RMAN)**

RMAN is an Oracle Database utility that integrates with an Oracle database to perform backup and recovery activities, including maintaining a repository of historical backup metadata in the control file of every database that it backs up. RMAN can also maintain a centralized backup repository called a **recovery catalog** in a different database. RMAN is an Oracle Database feature and does not require separate installation.

RMAN is integrated with **Oracle Secure Backup**, which provides reliable, centralized tape backup management, protecting file system data and Oracle Database files. The Oracle Secure Backup SBT interface enables you to use RMAN to back up and restore database files to and from tape and internet-based Web

Services such as Amazon S3. Oracle Secure Backup supports almost every tape drive and tape library in SAN and SCSI environments.

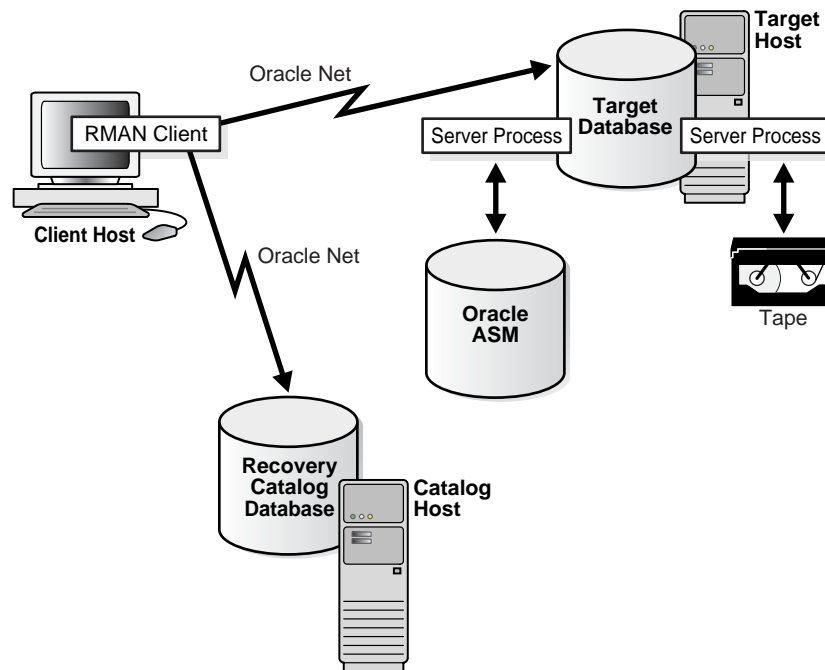
RMAN and Oracle Secure Backup are accessible both from the command line and from Enterprise Manager.

- User-Managed techniques

As an alternative to RMAN, you can use operating system commands such as the Linux `dd` for backing up and restoring files and the SQL*Plus `RECOVER` command for media recovery. User-managed backup and recovery is fully supported by Oracle, although RMAN is recommended because it is integrated with Oracle Database and simplifies administration.

Figure 18–3 shows basic RMAN architecture. The RMAN client, accessible through Enterprise Manager, uses server sessions on a target database to back up data to disk or tape. RMAN can update an external recovery catalog with backup metadata.

Figure 18–3 RMAN Architecture



Whichever backup and recovery technique you use, Oracle recommends that you configure a **fast recovery area**. This database-managed directory, file system, or **Oracle ASM disk group** centralizes backup and recovery files, including active control files, online and **archived redo log files**, and backups. Oracle Database recovery components interact with the fast recovery area to ensure database recoverability.

See Also:

- *Oracle Database 2 Day DBA* to learn how to perform backup and recovery with Enterprise Manager
- *Oracle Database Backup and Recovery User's Guide* for an overview of backup and recovery solutions
- *Oracle Database Administrator's Guide* for information about how to set up and administer the fast recovery area
- *Oracle Secure Backup Administrator's Guide* for an overview of Oracle Secure Backup

Database Backups

Database backups can be either physical or logical. **Physical backups**, which are the primary concern in a backup and recovery strategy, are copies of physical database files. You can make physical backups with RMAN or operating system utilities.

In contrast, **logical backups** contain logical data such as tables and stored procedures. You can extract logical data with an Oracle Database utility such as Data Pump Export and store it in a binary file. Logical backups can supplement physical backups.

Physical backups have large granularity and limited transportability, but are very fast. Logical backups have fine granularity and complete transportability, but are slower than physical backups.

See Also: *Oracle Database Backup and Recovery User's Guide* to learn about physical and logical backups

Whole and Partial Database Backups A **whole database backup** is a backup of every data file in the database, plus the control file. Whole database backups are the most common type of backup.

A **partial database backup** includes a subset of the database: individual tablespaces or data files. A **tablespace backup** is a backup of all the data files in a tablespace or in multiple tablespaces. Tablespace backups, whether consistent or inconsistent, are valid only if the database is operating in ARCHIVELOG mode because redo is required to make the restored tablespace consistent with the rest of the database.

Consistent and Inconsistent Backups A whole database backup is either consistent or inconsistent. In a **consistent backup**, all read/write data files and control files have the same **checkpoint SCN**, guaranteeing that these files contain all changes up to this SCN. This type of backup does not require recovery after it is restored.

A consistent backup of the database is only possible after a consistent shutdown (see "[Shutdown Modes](#)" on page 13-9) and is the only valid backup option for a database operating in NOARCHIVELOG mode. Other backup options require media recovery for consistency, which is not possible without applying archived redo log files.

Note: If you restore a consistent whole database backup without applying redo, then you lose all transactions made after the backup.

In an **inconsistent backup**, read/write data files and control files are not guaranteed to have the same checkpoint SCN, so changes can be missing. All online backups are necessarily inconsistent because data files can be modified while backups occur.

Inconsistent backups offer superior availability because you do not have to shut down the database to make backups that fully protect the database. If the database runs in ARCHIVELOG mode, and if you back up the archived redo logs and data files, then inconsistent backups can be the foundation for a sound backup and recovery strategy.

See Also: *Oracle Database Backup and Recovery User's Guide* to learn more about inconsistent backups

Backup Sets and Image Copies The RMAN BACKUP command generates either **image copies** or **backup sets**. An image copy is 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.

Note: Unlike operating system copies, RMAN validates the blocks in the file and records the image copy in the RMAN repository.

RMAN can also create backups in a proprietary format called a **backup set**. A backup set contains the data from one or more data files, archived redo log files, or control files or server parameter file. The smallest unit of a backup set is a binary file called a **backup piece**. Backup sets are the only form in which RMAN can write backups to sequential devices such as tape drives.

Backup sets enable tape devices to stream continuously. For example, RMAN can mingle blocks from slow, medium, and fast disks into one backup set so that the tape device has a constant input of blocks. Image copies are useful for disk because you can update them incrementally, and also recover them in place.

See Also: *Oracle Database Backup and Recovery User's Guide* to learn more about backup sets and image copies

Data Repair

While several problems can halt the normal operation of a database or affect I/O operations, only the following typically require DBA intervention and data repair:

- Media failures

A **media failure** occurs when a problem external to the database prevents it from reading from or writing to a file. Typical media failures include physical failures, such as head crashes, and the overwriting, deletion, or corruption of a database file. Media failures are less common than user or application errors, but a sound recovery strategy must prepare for them.

- User errors

A user or application may make unwanted changes to your database, such as erroneous updates, deleting the contents of a table, or dropping database objects (see "[Human Errors](#)" on page 17-9). A good backup and recovery strategy enables you to return your database to the desired state, with the minimum possible impact upon database availability, and minimal DBA effort.

Typically, you have multiple ways to solve the preceding problems. This section summarizes some of these solutions.

See Also: *Oracle Database 2 Day DBA* and *Oracle Database Backup and Recovery User's Guide* for data repair concepts

Data Recovery Advisor The **Data Recovery Advisor** tool automatically diagnoses persistent data failures, presents appropriate repair options, and executes repairs at the user's request. By providing a centralized tool for automated data repair, Data Recovery Advisor improves the manageability and reliability of an Oracle database and thus helps reduce recovery time.

The database includes a framework called **Health Monitor** for running diagnostic checks. A **checker** is a diagnostic operation or procedure registered with Health Monitor to assess the health of the database or its components. The health assessment is known as a **data integrity check** and can be invoked reactively or proactively.

A **failure** is a persistent data corruption detected by a data integrity check. Failures are normally detected reactively. A database operation involving corrupted data results in an error, which automatically invokes a data integrity check that searches the database for failures related to the error. If failures are diagnosed, then the database records them in the Automatic Diagnostic Repository (ADR).

After failures have been detected by the database and stored in ADR, Data Recovery Advisor automatically determines the best repair options and their impact on the database. Typically, Data Recovery Advisor generates both manual and automated repair options for each failure or group of failures.

Before presenting an automated repair option, Data Recovery Advisor validates it for the specific environment and for the availability of media components required to complete the proposed repair. If you choose an automatic repair, then Oracle Database executes it for you. The Data Recovery Advisor tool verifies the repair success and closes the appropriate failures.

See Also: *Oracle Database 2 Day DBA and Oracle Database Backup and Recovery User's Guide* to learn how to use Data Recovery Advisor

Oracle Flashback Technology Oracle Database provides a group of features known as **Oracle Flashback Technology** that support viewing past states of data, and winding data back and forth in time, without needing to restore backups. Depending on the database changes, flashback features can often reverse unwanted changes more quickly and with less impact on availability than media recovery.

The following flashback features are most relevant for backup and recovery:

- **Flashback Database**

You can rewind an Oracle database to a previous time to correct problems caused by logical data corruptions or user errors. Flashback Database can also be used to complement Data Guard, Data Recovery Advisor, and for synchronizing clone databases. Flashback Database does not restore or perform media recovery on files, so you cannot use it to correct media failures such as disk crashes.

- **Flashback Table**

You can rewind tables to a specified point in time with a single SQL statement. You can restore table data along with associated indexes, triggers, and constraints, while the database is online, undoing changes to only the specified tables. Flashback Table does not address physical corruption such as bad disks or data segment and index inconsistencies.

- **Flashback Drop**

You can reverse the effects of a `DROP TABLE` operation. Flashback Drop is substantially faster than recovery mechanisms such as point-in-time recovery and does not lead to loss of recent transactions or downtime.

See Also:

- *Oracle Database 2 Day DBA* and *Oracle Database Backup and Recovery User's Guide* to learn more about flashback features
- *Oracle Database SQL Language Reference* and *Oracle Database Backup and Recovery Reference* to learn about the `FLASHBACK DATABASE` statement

Block Media Recovery A **block corruption** is a data block that is not in a recognized Oracle format, or whose contents are not internally consistent (see "[Data Corruption](#)" on page 17-8). Block media recovery is a technique for restoring and recovering corrupt data blocks while data files are online. If only a few blocks are corrupt, then block recovery may be preferable to data file recovery.

See Also: *Oracle Database Backup and Recovery User's Guide* to learn how to perform block media recovery

Data File Recovery Data file recovery repairs a lost or damaged current data file or control file. It can also recover changes lost when a tablespace went offline without the `OFFLINE NORMAL` option.

Media recovery is necessary if you restore a backup of a data file or control file or a data file is taken offline without the `OFFLINE NORMAL` option. The database cannot be opened if online data files needs media recovery, nor can a data file that needs media recovery be brought online until media recovery completes.

To **restore** a physical backup of a data file or control file is to reconstruct it and make it available to Oracle Database. To **recover** a backup is to apply archived redo log files, thereby reconstructing lost changes. RMAN can also recover data files with **incremental backups**, which contain only blocks modified after a previous backup.

Unlike instance recovery, which automatically applies changes to online files, media recovery must be invoked by a user and applies archived redo log files to restored backups. Data file media recovery can only operate on offline data files or data files in a database that is not opened by any instance.

Data file media recovery differs depending on whether all changes are applied:

- **Complete recovery**
Complete recovery applies *all* redo changes contained in the archived and online logs to a backup. Typically, you perform complete media recovery after a media failure damages data files or the control file. You can perform complete recovery on a database, tablespace, or data file.
- **Incomplete recovery**
Incomplete recovery, also called **database point-in-time recovery**, results in a noncurrent version of the database. In this case, you do not apply all of the redo generated after the restored backup. Typically, you perform point-in-time database recovery to undo a user error when Flashback Database is not possible.

To perform incomplete recovery, you must restore all data files from backups created before the time to which you want to recover and then open the database with the `RESETLOGS` option when recovery completes. Resetting the logs creates a new stream of log sequence numbers starting with log sequence 1.

Note: If current data files are available, then Flashback Database is an alternative to DBPITR.

The **tablespace point-in-time recovery (TSPITR)** feature lets you recover one or more tablespaces to a point in time older than the rest of the database.

See Also:

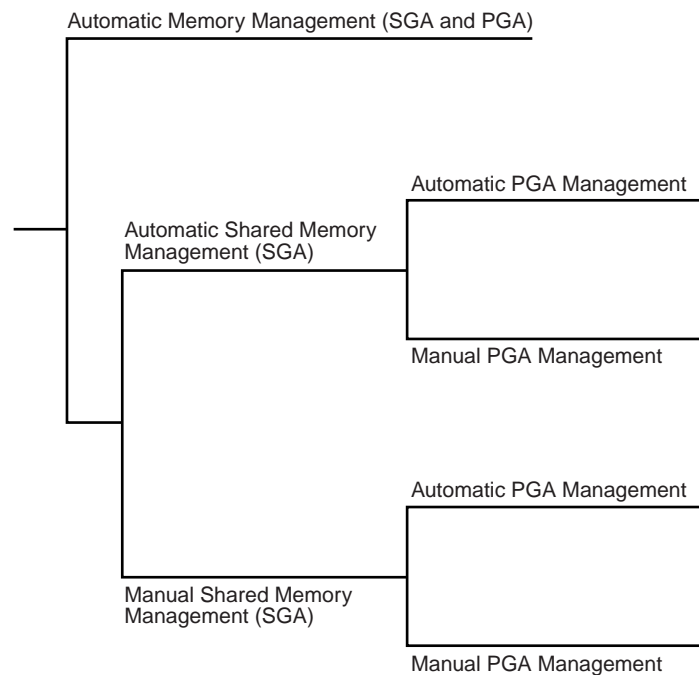
- ["Overview of Instance Recovery"](#) on page 13-12
- *Oracle Database 2 Day DBA* and *Oracle Database Backup and Recovery User's Guide* for media recovery concepts

Memory Management

Memory management involves maintaining optimal sizes for the Oracle instance memory structures as demands on the database change. Initialization parameter settings determine how SGA and instance PGA memory is managed.

[Figure 18–4](#) shows a decision tree for memory management options. The following sections explain the options in detail.

Figure 18–4 Memory Management Methods



See Also: [Chapter 14, "Memory Architecture"](#) to learn more about the SGA and PGA

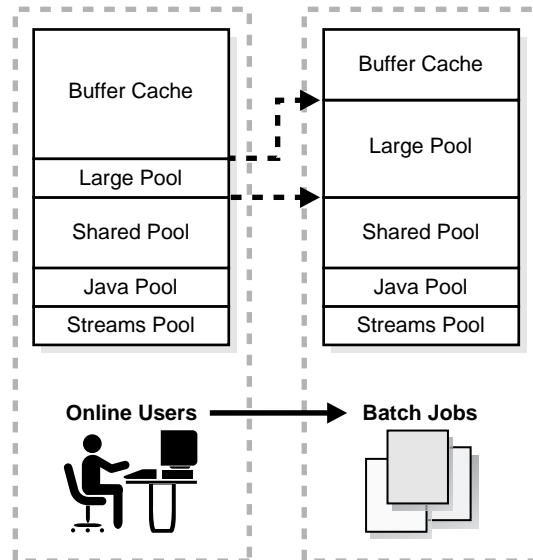
Automatic Memory Management

In **automatic memory management**, Oracle Database manages the SGA and instance PGA memory completely automatically. This method is the simplest and is strongly recommended by Oracle.

The only user-specified controls are the **target memory size** initialization parameter (MEMORY_TARGET) and optional **maximum memory size** initialization parameter (MEMORY_MAX_TARGET). Oracle Database tunes to the target memory size, redistributing memory as needed between the SGA and the instance PGA.

Figure 18–5 shows a database that sometimes processes jobs submitted by online users and sometimes batch jobs. Using automatic memory management, the database automatically adjusts the size of the **large pool** and **database buffer cache** depending on which type of jobs are running.

Figure 18–5 Automatic Memory Management



If you create your database with DBCA and choose the basic installation option, then automatic memory management is enabled by default.

See Also: *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn about automatic memory management

Shared Memory Management of the SGA

If automatic memory management is not enabled, then the system must use **shared memory management** of the SGA. Shared memory management is possible in either of the following forms:

- Automatic shared memory management

This mode enables you to exercise more direct control over the size of the SGA and is the default when automatic memory management is disabled. The database tunes the total SGA to the target size and dynamically tunes the sizes of SGA components. Oracle Database remembers the sizes of the automatically tuned components across instance shutdowns if you are using a server parameter file.

- Manual shared memory management

In this mode, you set the sizes of several individual SGA components and manually tune individual SGA components on an ongoing basis. You have complete control of individual SGA component sizes. The database defaults to this mode when both automatic memory management and automatic shared memory management are disabled.

See Also: *Oracle Database 2 Day DBA* and *Oracle Database Administrator's Guide* to learn about shared memory management

Memory Management of the Instance PGA

If automatic memory management is not enabled, then the following modes are possible for management of PGA memory:

- Automatic PGA memory management

When automatic memory management is disabled and `PGA_AGGREGATE_TARGET` is set to a nonzero value, the database uses **automatic PGA memory management**. In this mode, the `PGA_AGGREGATE_TARGET` specifies a target size for the instance PGA. The database then tunes the size of the instance PGA to this target and dynamically tunes the sizes of individual PGAs. If you do not explicitly set a target size, then the database automatically configures a reasonable default.

- Manual PGA memory management

When automatic memory management is disabled and `PGA_AGGREGATE_TARGET` is set to 0, the database defaults to **manual PGA management**. Previous releases of Oracle Database required the DBA to manually specify the maximum work area size for each type of SQL **operator** (such as a sort or **hash join**). This technique proved to be very difficult because the workload is always changing. Although Oracle Database supports the manual PGA memory management method, Oracle strongly recommends automatic memory management.

See Also: *Oracle Database Performance Tuning Guide* to learn about PGA memory management

Summary of Memory Management Methods

[Table 18-1](#) summarizes the various memory management methods. If you do not enable automatic memory management, then you must separately configure one memory management method for the SGA and one for the PGA.

Note: When automatic memory management is not enabled, the default method for the instance PGA is automatic PGA memory management.

Table 18–1 Memory Management Methods

Instance	SGA	PGA	Description	Initialization Parameters
Auto	n/a	n/a	The database tunes the size of the instance based on a single instance target size.	You set: <ul style="list-style-type: none"> Total memory target size for the database instance (<code>MEMORY_TARGET</code>) Optional maximum memory size for the database instance (<code>MEMORY_MAX_TARGET</code>)
n/a	Auto	Auto	The database automatically tunes the SGA based on an SGA target. The database automatically tunes the PGA based on a PGA target.	You set: <ul style="list-style-type: none"> SGA target size (<code>SGA_TARGET</code>) Optional SGA maximum size (<code>SGA_MAX_SIZE</code>) Instance PGA target size (<code>PGA_AGGREGATE_TARGET</code>)
n/a	Auto	Manual	The database automatically tunes the SGA based on an SGA target. You control the PGA manually, setting the maximum work area size for each type of SQL operator.	You set: <ul style="list-style-type: none"> SGA target size (<code>SGA_TARGET</code>) Optional SGA maximum size (<code>SGA_MAX_SIZE</code>) PGA work area parameters such as <code>SORT_AREA_SIZE</code>, <code>HASH_AREA_SIZE</code>, and <code>BITMAP_MERGE_AREA_SIZE</code>
n/a	Manual	Auto	You control the SGA manually by setting individual component sizes. The database automatically tunes the PGA based on a PGA target.	You set: <ul style="list-style-type: none"> Shared pool size (<code>SHARED_POOL_SIZE</code>) Buffer cache size (<code>DB_CACHE_SIZE</code>) Large pool size (<code>LARGE_POOL_SIZE</code>) Java pool size (<code>JAVA_POOL_SIZE</code>) Streams pool size (<code>STREAMS_POOL_SIZE</code>) Instance PGA target size (<code>PGA_AGGREGATE_TARGET</code>)
n/a	Manual	Manual	You must manually configure SGA component sizes. You control the PGA manually, setting the maximum work area size for each type of SQL operator.	You must manually configure SGA component sizes. You set: <ul style="list-style-type: none"> Shared pool size (<code>SHARED_POOL_SIZE</code>) Buffer cache size (<code>DB_CACHE_SIZE</code>) Large pool size (<code>LARGE_POOL_SIZE</code>) Java pool size (<code>JAVA_POOL_SIZE</code>) Streams pool size (<code>STREAMS_POOL_SIZE</code>) PGA work area parameters such as <code>SORT_AREA_SIZE</code>, <code>HASH_AREA_SIZE</code>, and <code>BITMAP_MERGE_AREA_SIZE</code>

See Also: *Oracle Database Administrator's Guide* because automatic memory management is not available on all platforms

Resource Management and Task Scheduling

In a database with many active users, resource management is an important part of database administration. Sessions that consume excessive resources can prevent other sessions from doing their work. A related problem is how to schedule tasks so that they run at the best time. Oracle Database provides tools to help solve these problems.

Database Resource Manager

Oracle Database Resource Manager (the Resource Manager) is an infrastructure that provides granular control of database resources allocated to users, applications, and services. The Resource Manager solves many resource allocation problems that an operating system does not manage well, including:

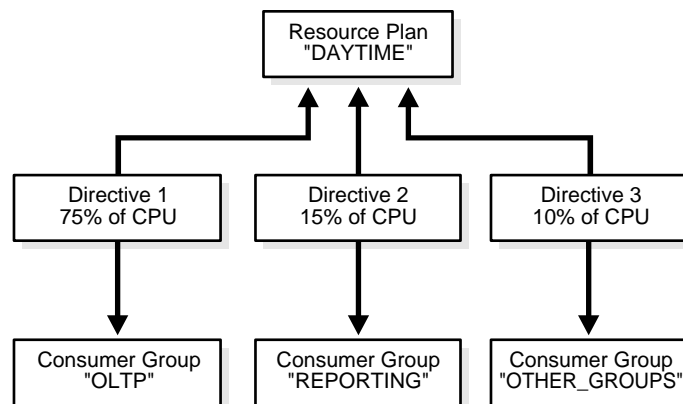
- Excessive overhead
- Inefficient scheduling
- Inappropriate allocation of resources
- Inability to manage database-specific resources

The Resource Manager helps overcome these problems by giving the database more control over allocation of hardware resources and enabling you to prioritize work within the database. You can classify sessions into groups based on session attributes, and then allocate resources to these groups to optimize hardware utilization.

Resources are allocated to users according to a **resource plan** specified by the database administrator. The plan specifies how the resources are to be distributed among **resource consumer groups**, which are user sessions grouped by resource requirements. A **resource plan directive** associates a resource consumer group with a plan and specifies how resources are to be allocated to the group.

Figure 18–6 shows a simple resource plan for an organization that runs **OLTP** applications and reporting applications simultaneously during the daytime. The currently active plan, **DAYTIME**, allocates CPU resources among three resource consumer groups. Specifically, **OLTP** is allotted 75% of the CPU time, **REPORTS** is allotted 15%, and **OTHER_GROUPS** receives the remaining 10%.

Figure 18–6 Simple Resource Plan



See Also: *Oracle Database Administrator's Guide* for information about using the Resource Manager

Oracle Scheduler

Oracle Scheduler (the Scheduler) enables database administrators and application developers to control when and where various tasks take place in the database environment. The Scheduler provides complex enterprise scheduling functionality, which you can use to:

- Schedule job execution based on time or events
- Schedule job processing in a way that models your business requirements

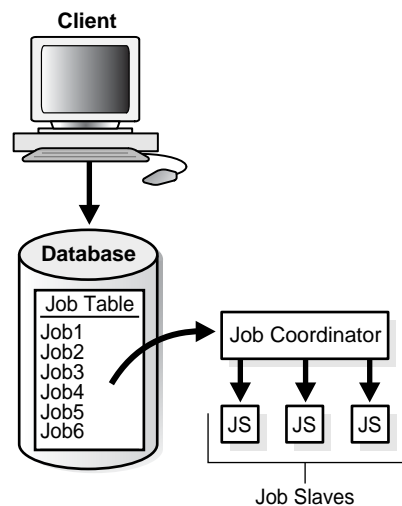
- Manage and monitor jobs
- Execute and manage jobs in a clustered environment

Program objects (**programs**) contain metadata about the command that the Scheduler will run, including default values for any arguments. Schedule objects (**schedules**) contain information about run date and time and recurrence patterns. Job objects (**jobs**) associate a program with a schedule. To define what is executed and when, you assign relationships among programs, schedules, and jobs.

The Scheduler is implemented as a set of functions and procedures in the `DBMS_SCHEDULER` PL/SQL package. You create and manipulate Scheduler objects with this package or with Enterprise Manager. Because Scheduler objects are standard database objects, you can control access to them with system and object privileges.

Figure 18–7 shows the basic architecture of the Scheduler. The job table is a container for all the jobs, with one table per database. The **job coordinator background process** is automatically started and stopped as needed. Job slaves are awakened by the coordinator when a job must be run (see "Job Queue Processes (CJQO and Jnnn)" on page 15-12). The slaves gather metadata from the job table and run the job.

Figure 18–7 Scheduler Components



See Also: *Oracle Database Administrator's Guide* to learn about the Scheduler

Performance Diagnostics and Tuning

As a DBA, you are responsible for the performance of your Oracle database. Typically, performance problems result from unacceptable **response time**, which is the time to complete a specified workload, or **throughput**, which is the amount of work that can be completed in a specified time. Common problems include:

- CPU bottlenecks
- Undersized memory structures
- I/O capacity issues
- Inefficient or high-load SQL statements
- Unexpected performance regression after tuning SQL statements
- Concurrency and contention issues

- Database configuration issues

The general goal of tuning is usually to improve response time, increase throughput, or both. A specific and measurable goal might be "Reduce the response time of the specified `SELECT` statement to under 5 seconds." Whether this goal is achievable depends on factors that may or may not be under the control of the DBA. In general, tuning is the effort to achieve specific, measurable, and achievable tuning goals by using database resources in the most efficient way possible.

The **Oracle performance method** is based on identifying and eliminating bottlenecks in the database, and developing efficient SQL statements. Applying the Oracle performance method involves the following tasks:

- Performing pre-tuning preparations
- Tuning the database proactively on a regular basis
- Tuning the database reactively when users report performance problems
- Identifying, tuning, and optimizing high-load SQL statements

This section describes essential aspects of Oracle Database performance tuning, including the use of **advisors**. Oracle Database advisors provide specific advice on how to address key database management challenges, covering a wide range of areas including space, performance, and undo management.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database Performance Tuning Guide* provide to learn how to implement the Oracle performance method

Database Self-Monitoring

Self-monitoring take place as the database performs its regular operation, ensuring that the database is aware of problems as they arise. Oracle Database can send a **server-generated alert** to notify you of an impending problem.

Alerts are automatically generated when a problem occurs or when data does not match expected values for metrics such as physical reads per second or SQL response time. A **metric** is the rate of change in a cumulative statistic. Server-generated alerts can be based on user-specified threshold levels or because an event has occurred.

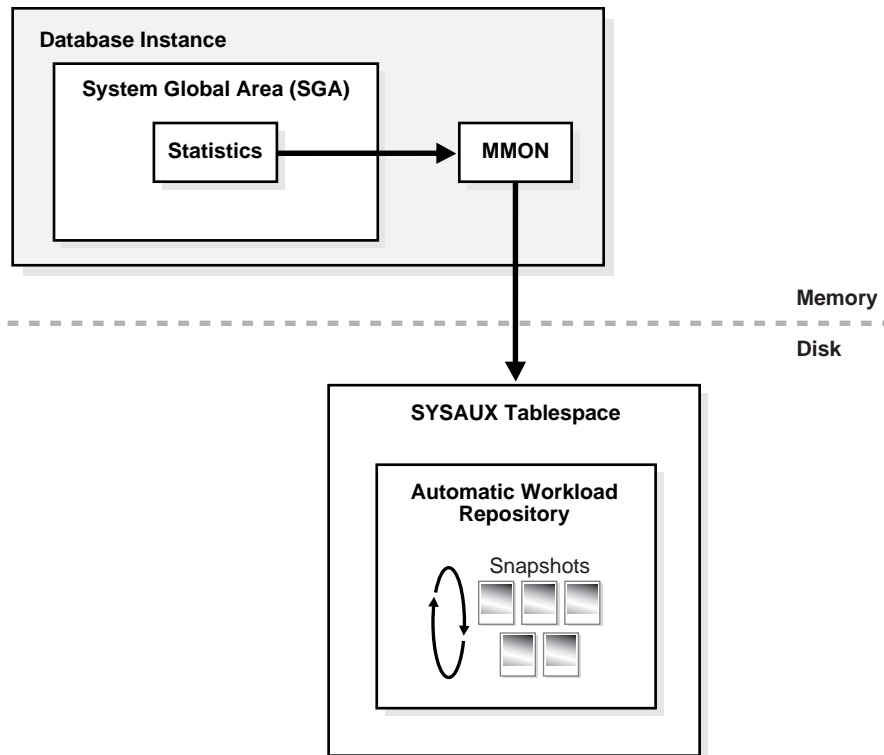
Server-generated alerts not only identify the problem, but sometimes recommend how the reported problem can be resolved. An example is an alert that the fast recovery area is running out of space with the recommendation that obsolete backups should be deleted or additional disk space added.

See Also: *Oracle Database Administrator's Guide*

Automatic Workload Repository (AWR)

Automatic Workload Repository (AWR) is a repository of historical performance data that includes cumulative statistics for the system, sessions, individual SQL statements, segments, and services. These statistics are the foundation of performance tuning. By automating the gathering of database statistics for problem detection and tuning, AWR serves as the foundation for database self-management.

As shown in [Figure 18–8](#), the database stores recent AWR statistics in the SGA. By default, the MMON process gathers statistics every hour and creates an **AWR snapshot** (see "[Manageability Monitor Processes \(MMON and MMNL\)](#)" on page 15-11). A snapshot is a set of performance statistics captured at a specific time. The database writes snapshots to the `SYS_AUX` tablespace. AWR manages snapshot space, purging older snapshots according to a configurable **snapshot retention policy**.

Figure 18–8 Automatic Workload Repository (AWR)

An **AWR baseline** is a collection of statistic rates usually taken over a period when the system is performing well at peak load. You can specify a pair or range of AWR snapshots as a baseline. By using an AWR report to compare statistics captured during a period of bad performance to a baseline, you can diagnose problems.

An automated maintenance infrastructure known as **AutoTask** illustrates how Oracle Database uses AWR for self-management. By analyzing AWR data, AutoTask can determine the need for maintenance tasks and schedule them to run in Oracle Scheduler **maintenance windows**. Examples of tasks include gathering statistics for the **optimizer** and running the Automatic Segment Advisor.

See Also:

- ["The SYSAUX Tablespace"](#) on page 12-32
- *Oracle Database Performance Tuning Guide* to learn about AWR
- *Oracle Database Administrator's Guide* and *Oracle Database 2 Day DBA* to learn how to manage automatic maintenance tasks

Automatic Database Diagnostic Monitor (ADDM)

Automatic Database Diagnostic Monitor (ADDM) is a self-diagnostic advisor built into Oracle Database. Using statistics captured in AWR, ADDM automatically and proactively diagnoses database performance and determines how identified problems can be resolved. You can also run ADDM manually.

ADDM takes a holistic approach to system performance, using time as a common currency between components. ADDM identifies areas of Oracle Database consuming the most time. For example, the database may be spending an excessive amount of time waiting for free database buffers. ADDM drills down to identify the root cause of

problems, rather than just the symptoms, and reports the effect of the problem on Oracle Database overall. Minimal overhead occurs during the diagnostic process.

In many cases, ADDM recommends solutions and quantifies expected performance benefits. For example, ADDM may recommend changes to hardware, database configuration, database schema, or applications. If a recommendation is made, then ADDM reports the time benefit. The use of time as a measure enables comparisons of problems or recommendations.

Besides reporting potential performance issues, ADDM documents areas of the database that are not problems. Subcomponents such as I/O and memory that are not significantly impacting database performance are pruned from the classification tree at an early stage. ADDM lists these subcomponents so that you can quickly see that there is little benefit to performing actions in those areas.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database Performance Tuning Guide*

Active Session History (ASH)

Active Session History (ASH) samples active database sessions each second, writing the data to memory and persistent storage. ASH is an integral part of the database self-management framework and is useful for diagnosing performance problems.

Unlike instance-level statistics gathered by AWR, ASH statistics are gathered at the session level. An **active session** is a session that is using CPU and is not waiting for an event in the idle wait class.

You can use Enterprise Manager or SQL scripts to generate **ASH reports** that gather session statistics gathered over a specified duration. You can use ASH reports for:

- Analysis of short-lived performance problems not identified by ADDM
- Scoped or targeted performance analysis by various dimensions or their combinations, such as time, session, module, action, or SQL ID

For example, a user notifies you that the database was slow between 10:00 p.m. and 10:02 p.m. However, the 2-minute performance degradation represents a small portion of the AWR snapshot interval from 10:00 p.m. and 11:00 p.m. and does not appear in ADDM findings. ASH reports can help identify the source of the transient problem.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database Performance Tuning Guide*

Application and SQL Tuning

Oracle Database completely automates the SQL tuning process. ADDM identifies SQL statements consuming unusually high system resources and therefore causing performance problems. In addition, AWR automatically captures the top SQL statements in terms of CPU and shared memory consumption. The identification of high-load SQL statements happens automatically and requires no intervention.

SQL Tuning Advisor Automatic SQL tuning is exposed through **SQL Tuning Advisor**. SQL Tuning Advisor runs automatically during system maintenance windows as a maintenance task. During each automatic run, the advisor selects high-load SQL queries in the database and generates recommendations for tuning these queries.

SQL Tuning Advisor recommendations fall into the following categories:

- Statistics analysis
- SQL profiling

- Access path analysis
- SQL structure analysis

A **SQL profile** contains additional statistics specific to a SQL statement and enables the optimizer to generate a better **execution plan**. Essentially, a SQL profile is a method for analyzing a query. Both **access path** and SQL structure analysis are useful for tuning an application under development or a homegrown production application.

A principal benefit of SQL Tuning Advisor is that solutions come from the optimizer rather than external tools (see "[Overview of the Optimizer](#)" on page 7-10). Thus, tuning is performed by the database component that is responsible for the execution plans and SQL performance. The tuning process can consider past execution statistics of a SQL statement and customizes the optimizer settings for this statement.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database Performance Tuning Guide*

SQL Access Advisor **SQL Access Advisor** offers advice on how to optimize data access paths. Specifically, it recommends how database performance can be improved through partitioning, materialized views, indexes, and materialized view logs.

Schema objects such as partitions and indexes are essential for optimizing complex, data-intensive queries. However, creation and maintenance of these objects can be time-consuming, and space requirements can be significant. SQL Access Advisor helps meet performance goals by recommending data structures for a specified workload.

The SQL Access Advisor can be run from Enterprise Manager using the SQL Access Advisor Wizard or by invoking the DBMS_ADVISOR package. The DBMS_ADVISOR package consists of a collection of analysis and advisory functions and procedures callable from any PL/SQL program.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* and *Oracle Database Performance Tuning Guide*

Concepts for Database Developers

The Oracle Database developer creates and maintains a database application. This section presents a brief overview of what a database developer does and the development tools available.

This section contains the following topics:

- [Duties of Database Developers](#)
- [Tools for Database Developers](#)
- [Topics for Database Developers](#)

Duties of Database Developers

An Oracle developer is responsible for creating or maintaining the database components of an application that uses the Oracle technology stack. Oracle developers either develop new applications or convert existing applications to run in an Oracle Database environment. For this reason, developers work closely with the database administrators, sharing knowledge and information.

Oracle database developers can expect to be involved in the following tasks:

- Implementing the data model required by the application
- Creating [schema objects](#) and implementing rules for [data integrity](#)
- Choosing a programming environment for a new development project
- Writing server-side [PL/SQL](#) or Java subprograms and client-side procedural code that use [SQL](#) statements
- Creating the application interface with the chosen development tool
- Establishing a Globalization Support environment for developing globalized applications
- Instantiating applications in different databases for development, testing, education, and deployment in a production environment

The preceding tasks, and many others, are described in *Oracle Database 2 Day Developer's Guide* and *Oracle Database Advanced Application Developer's Guide*.

See Also: ["Introduction to Server-Side Programming"](#) on page 8-1

Tools for Database Developers

Oracle provides several tools for use in developing database applications. This section describes some commonly used development tools.

SQL Developer

SQL Developer is a graphical version of **SQL*Plus**, written in Java, that supports development in SQL and PL/SQL. You can connect to any Oracle database schema using standard database authentication. SQL Developer enables you to:

- Browse, create, edit, and delete schema objects
- Execute SQL statements
- Edit and debug PL/SQL program units
- Manipulate and export data
- Create and display reports

SQL Developer is available in the default Oracle Database installation and by free download.

See Also: *Oracle Database 2 Day Developer's Guide* and *Oracle Database SQL Developer User's Guide* to learn how to use SQL Developer

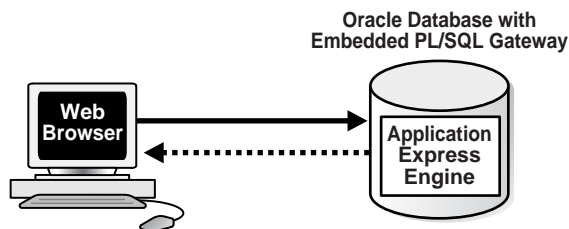
Oracle Application Express

Oracle Application Express (APEX) is a Web application development tool for Oracle Database. The tool uses built-in features such as user interface themes, navigational controls, form handlers, and flexible reports to accelerate application development.

Oracle Application Express installs with the database and consists of data in tables and PL/SQL code. When you run an application, your browser sends a URL request that is translated into an Oracle Application Express PL/SQL call. After the database processes the PL/SQL, the results are relayed back to the browser as HTML. This cycle happens each time you request or submit a page.

You can use Oracle Application Express with the embedded PL/SQL gateway. The gateway runs in the XML DB HTTP server in the database and provides the necessary infrastructure to create dynamic applications. As shown in [Figure 19–1](#), the embedded PL/SQL gateway simplifies the application architecture by eliminating the middle tier.

Figure 19–1 Application Express with Embedded PL/SQL Gateway



See Also: *Oracle Database 2 Day + Application Express Developer's Guide* to learn how to use APEX

Oracle JDeveloper

Oracle JDeveloper is an integrated development environment (IDE) for building service-oriented applications using the latest industry standards for Java, XML, Web services, and SQL. Oracle JDeveloper supports the complete software development life cycle, with integrated features for modeling, coding, debugging, testing, profiling, tuning, and deploying applications.

JDeveloper uses windows for various application development tools. For example, when creating a Java application, you can use tools such as the Java Visual Editor and Component Palette. In addition to these tools, JDeveloper provides a range of navigators to help you organize and view the contents of your projects.

See Also:

- *Oracle Database 2 Day + Java Developer's Guide* to learn how to use JDeveloper
- You can download JDeveloper from the following URL:
<http://www.oracle.com/technetwork/developer-tools/jdev/downloads/>

Oracle JPublisher

Java Publisher (JPublisher) is a simple and convenient tool to create Java programs that access database tables. Java code stubs generated by JDeveloper present object-relational structures in the database as Java classes. These classes can represent the following user-defined database entities in a Java program:

- SQL object types
- Object reference types
- SQL collection types
- PL/SQL packages

You can specify and customize the mapping of these entities to Java classes in a **strongly typed** paradigm, so that a specific Java type is associated with a specific user-defined SQL type. JPublisher can also generate classes for PL/SQL packages. These classes have wrapper methods to call the **stored procedure** in the package.

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

Oracle Developer Tools for Visual Studio .NET

Oracle Developer Tools for Visual Studio .NET is 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 Developer Tools support the programming and implementation of .NET stored procedures using Visual Basic, C#, and other .NET languages. These procedures are written in a .NET language and contain SQL or PL/SQL statements.

See Also: *Oracle Database 2 Day + .NET Developer's Guide for Microsoft Windows*

Topics for Database Developers

[Chapter 17](#) describes topics important for both developers and administrators. This section covers topics that are most essential to database developers and that have not been discussed elsewhere in the manual.

This section contains the following topics:

- [Principles of Application Design and Tuning](#)
- [Client-Side Database Programming](#)

- [Globalization Support](#)
- [Unstructured Data](#)

Principles of Application Design and Tuning

Oracle developers must design, create, and tune database applications so that they achieve security and performance goals. The following principles of application design and tuning are useful guidelines:

- Understand how Oracle Database works

As a developer, you want to develop applications in the least amount of time against an Oracle database, which requires exploiting the database architecture and features. For example, not understanding Oracle Database **concurrency** controls and multiversioning **read consistency** may make an application corrupt the integrity of the data, run slowly, and decrease scalability (see "[Introduction to Data Concurrency and Consistency](#)" on page 9-1).
- Use bind variables

When a **query** uses **bind variables**, the database can compile it once and store the **query plan** in the **shared pool**. If the same statement is executed again, then the database can perform a **soft parse** and reuse the plan. In contrast, a **hard parse** takes longer and uses more resources (see "[SQL Parsing](#)" on page 7-16). Using bind variables to allow soft parsing is very efficient and is the way the database intends developers to work.
- Implement **integrity constraints** in the server rather than in the client

Using primary and foreign keys enables data to be reused in multiple applications. Coding the rules in a client means that other clients do not have access to these rules when running against the databases (see "[Advantages of Integrity Constraints](#)" on page 5-1).
- Build a test environment with representative data and session activity

A test environment that simulates your live production environment provides multiple benefits. For example, you can benchmark the application to ensure that it scales and performs well. Also, you can use a test environment to measure the performance impact of changes to the database, and ensure that upgrades and patches work correctly.
- Design the data model with the goal of good performance

Typically, attempts to use generic data models result in poor performance. A well-designed data model answer the most common queries as efficiently as possible. For example, the data model should use the type of indexes that provide the best performance. Tuning after deployment is undesirable because changes to logic and physical structures may be difficult or impossible.
- Define clear performance goals and keep historical records of metrics

An important facet of development is determining exactly how the application is expected to perform and scale. For example, you should use metrics that include expected user load, transactions per second, acceptable response times, and so on. Good practice dictates that you maintain historical records of performance metrics. In this way, you can monitor performance proactively and reactively (see "[Performance Diagnostics and Tuning](#)" on page 18-20).
- Instrument the application code

Good development practice involves adding debugging code to your application. The ability to generate trace files is useful for debugging and diagnosing performance problems.

See Also: *Oracle Database 2 Day Developer's Guide* for considerations when designing and deploying database applications

Client-Side Database Programming

As explained in [Chapter 8, "Server-Side Programming: PL/SQL and Java"](#), two basic techniques enable procedural database applications to use SQL: server-side programming with PL/SQL and Java, and client-side programming with precompilers and APIs. This section provides a brief overview of client-side database programming.

See Also: *Oracle Database Advanced Application Developer's Guide* to learn how to choose a programming environment

Embedded SQL

Historically, client/server programs have used **embedded SQL** to interact with the database. This section explains options for using embedded SQL.

Oracle Precompilers Client/server programs are typically written using **precompilers**, which are programming tools that enable you to embed SQL statements in high-level programs. For example, the Oracle Pro*C/C++ precompiler enables you to embed SQL statements in a C or C++ source file. Oracle precompilers are also available for COBOL and FORTRAN.

A precompiler provides several benefits, including the following:

- Increases productivity because you typically write less code than equivalent OCI applications
- Enables you to create highly customized applications
- Allows close monitoring of resource use, SQL statement execution, and various run-time indicators
- Saves time because the precompiler, not you, translates each embedded SQL statement into calls to the Oracle Database run-time library
- Uses the Object Type Translator to map Oracle Database object types and collections into C data types to be used in the Pro*C/C++ application
- Provides compile time type checking of object types and collections and automatic type conversion from database types to C data types

The client application containing the SQL statements is the **host program**. This program is written in the **host language**. In the host program, you can mix complete SQL statements with complete C statements and use C variables or structures in SQL statements. When embedding SQL statements you must begin them with the keywords `EXEC SQL` and end them with a semicolon. Pro*C/C++ translates `EXEC SQL` statements into calls to the run-time library `SQLLIB`.

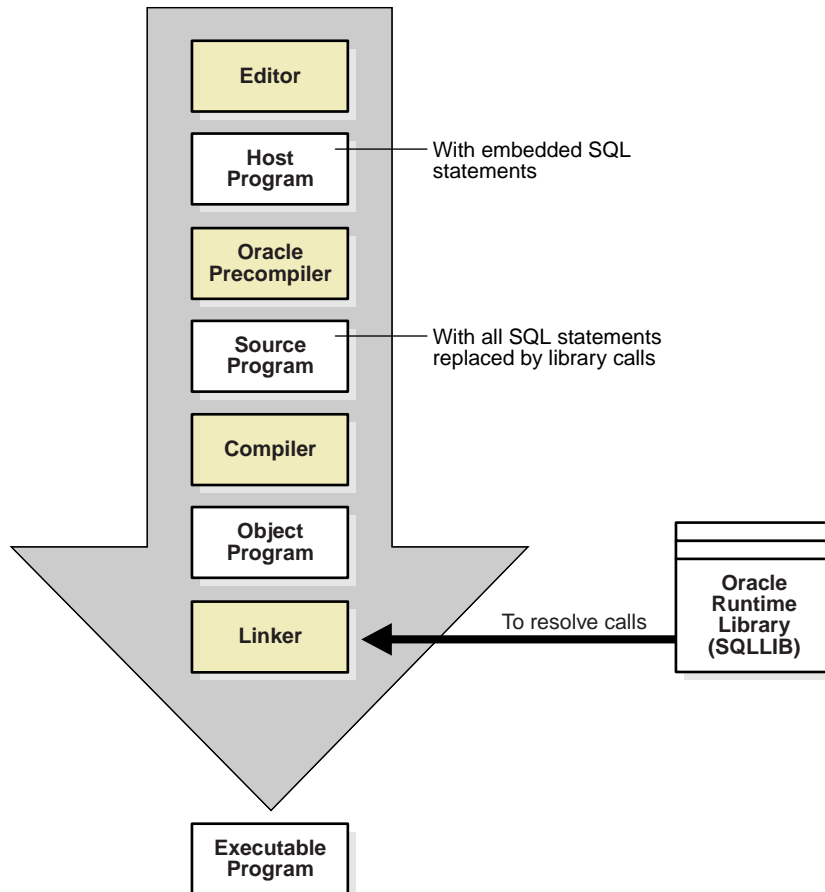
Many embedded SQL statements differ from their interactive counterparts only through the addition of a new clause or the use of program variables. The following example compares interactive and embedded `ROLLBACK` statements:

```
ROLLBACK;           -- interactive
EXEC SQL ROLLBACK; -- embedded
```

The statements have the same effect, but you would use the first in an interactive SQL environment (such as SQL Developer), and the second in a Pro*C/C++ program.

A precompiler accepts the host program as input, translates the embedded SQL statements into standard database run-time library calls, and generates a source program that you can compile, link, and run in the usual way. Figure 19–2 illustrates typical steps of developing programs that use precompilers.

Figure 19–2 Program Development with Precompilers



See Also:

- *Pro*C/C++ Programmer's Guide* for a complete description of the Pro*C/C++ precompiler
- *Pro*FORTRAN Supplement to the Oracle Precompilers Guide*

SQLJ SQLJ is an ANSI SQL-1999 standard for embedding SQL statements in Java source code. SQLJ provides a simpler alternative to the Java Database Connectivity (JDBC) API for client-side SQL data access from Java.

The SQLJ interface is the Java equivalent of the Pro* interfaces. You insert SQL statements in your Java source code. Afterward, you submit the Java source files to the SQLJ translator, which translates the embedded SQL to pure JDBC-based Java code.

See Also: "SQLJ" on page 8-16

Client-Side APIs

Most developers today use an API to embed SQL in their database applications. For example, two popular APIs for enabling programs to communicate with Oracle Database are Open Database Connectivity (ODBC) and JDBC. The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI) are two other common APIs for client-side programming.

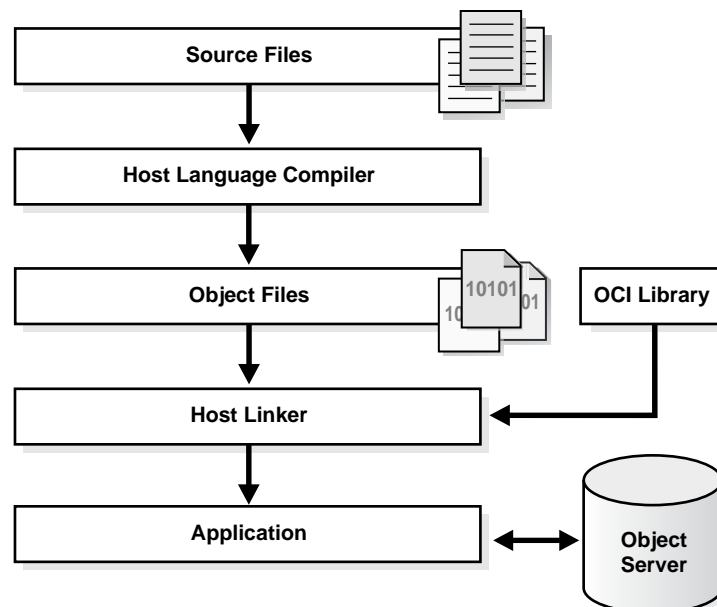
OCI and OCCI As an alternative to precompilers, Oracle provides the OCI and OCCI APIs. OCI lets you manipulate data and schemas in a database using a host programming language such as C. OCCI is an object-oriented interface suitable for use with C++. Both APIs enable developers to use native subprogram invocations to access Oracle Database and control SQL execution.

In some cases, OCI provides better performance or more features than higher-level interfaces. OCI and OCCI provide many features, including the following:

- Support for all SQL **DDL**, **DML**, query, and transaction control facilities available through Oracle Database
- Instant client, a way to deploy applications when disk space is an issue
- Thread management, connection pooling, globalization functions, and direct path loading of data from a C application

OCI and OCCI provide a library of standard database access and retrieval functions in the form of a dynamic run-time library (OCILIB). This library can be linked in an application at run time. Thus, you can compile and link an OCI or OCCI program in the same way as a nondatabase application, avoiding a separate preprocessing or precompilation step. [Figure 19-3](#) illustrates the development process.

Figure 19-3 Development Process Using OCI or OCCI



See Also:

- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*

ODBC and JDBC ODBC is a standard API that enables applications to connect to a database and then prepare and run SQL statements. ODBC is independent of programming language, database, and operating system. The goal of ODBC is to enable any application to access data contained in any database.

A **database driver** is software that sits between an application and the 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.

The Oracle ODBC driver provided by Oracle enables ODBC-compliant applications to access Oracle Database. For example, an application written in Visual Basic can use ODBC to query and update tables in an Oracle database.

JDBC is a low-level Java interface that enables Java applications to interact with Oracle database. Like ODBC, JDBC is a vendor-independent API. The JDBC standard is defined by Sun Microsystems and implemented through the `java.sql` interfaces.

The JDBC standard enables individual providers to implement and extend the standard with their own JDBC drivers. Oracle provides the following JDBC drivers for client-side programming:

- **JDBC thin driver**

This pure Java driver resides on the client side without an Oracle client installation. It is platform-independent and usable with both applets and applications.

- **JDBC OCI driver**

This driver resides on the client-side with an Oracle client installation. It is usable only with applications. The JDBC OCI driver, which is written in both C and Java, converts JDBC calls to OCI calls.

The following snippets are from a Java program that uses the JDBC OCI driver to create a `Statement` object and query the `dual` table:

```
// Create a statement
Statement stmt = conn.createStatement();

// Query dual table
ResultSet rset = stmt.executeQuery("SELECT 'Hello World' FROM DUAL");
```

See Also:

- *Oracle Database Advanced Application Developer's Guide* and *Oracle Database 2 Day + Java Developer's Guide* to learn more about JDBC
- *Oracle Database Gateway for ODBC User's Guide*

Globalization Support

Oracle Database **globalization support** enables you to store, process, and retrieve data in native languages. Thus, you can develop multilingual applications and software that can be accessed and run from anywhere in the world simultaneously.

Developers who write globalized database application must do the following:

- Understand the Oracle Database globalization support architecture, including the properties of the different character sets, territories, languages, and linguistic sort definitions

- Understand the globalization functionality of their middle-tier programming environment, including how it can interact and synchronize with the locale model of the database
- Design and write code capable of simultaneously supporting multiple clients running on different operating systems, with different character sets and locale requirements

For example, an application may be required to render content of the user interface and process data in languages and locale preferences of native users. For example, the application must process multibyte Kanji data, display messages and dates in the proper regional format, and process 7-bit ASCII data without requiring users to change settings.

See Also: *Oracle Database Globalization Support Guide* for more information about globalization

Globalization Support Environment

The **globalization support environment** includes the client application and the database. You can control language-dependent operations by setting parameters and environment variables on the client and server, which may exist in separate locations.

Note: In previous releases, Oracle referred to globalization support capabilities as National Language Support (NLS) features. NLS is actually a subset of globalization support and provides the ability to choose a national language and store data in a specific character set.

Oracle Database provides globalization support for features such as:

- Native languages and territories
- Local formats for date, time, numbers, and currency
- Calendar systems (Gregorian, Japanese, Imperial, Thai Buddha, and so on)
- Multiple character sets, including Unicode
- Character semantics

Character Sets A key component of globalization support is a **character set**, which is an encoding scheme used to display characters on your computer screen. The following distinction is important in application development:

- A **database character set** determines which languages can be represented in a database. The character set is specified at database creation.

Note: After a database is created, changing its character set is usually very expensive in terms of time and resources. This operation may require converting all character data by exporting the whole database and importing it back.

- A **client character set** is the character set for data entered or displayed by a client application. The character set for the client and database can be different.

A group of characters (for example, alphabetic characters, ideographs, symbols, punctuation marks, and control characters) can be encoded as a character set. An **encoded character set** assigns a unique numeric code, called a **code point** or **encoded**

value, to each character in the set. Code points are important in a global environment because of the potential need to convert data between different character sets.

The computer industry uses many encoded character sets. These sets differ in the number of characters available, the characters available for use, code points assigned to each character, and so on. Oracle Database supports most national, international, and vendor-specific encoded character set standards.

Oracle Database supports the following classes of encoded character sets:

- **Single-Byte character sets**
Each character occupies one byte. An example of a 7-bit character set is US7ASCII. An example of an 8-bit character set is WE8DEC.
- **Multibyte character sets**
Each character occupies multiple bytes. Multibyte sets are commonly used for Asian languages.
- **Unicode**
The universal encoded character set enables you to store information in any language by using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language.

See Also: *Oracle Database Globalization Support Guide* to learn about character set migration

Locale-Specific Settings A **locale** is a linguistic and cultural environment in which a system or program is running. **NLS parameters** determine locale-specific behavior on both the client and database. A database session uses NLS settings when executing statements on behalf of a client. For example, the database makes the correct territory usage of the thousands separator for a client.

Typically, the `NLS_LANG` environment variable on the client host specifies the locale for both the server session and client application. The process is as follows:

1. When a client application starts, it initializes the client NLS environment from the environment settings.
All NLS operations performed locally, such as displaying formatting in Oracle Developer applications, use these settings.
2. The client communicates the information defined by `NLS_LANG` to the database when it connects.
3. The database session initializes its NLS environment based on the settings communicated by the client.

If the client did not specify settings, then the session uses the settings in the initialization parameter file. The database uses the initialization parameter settings only if the client did not specify any NLS settings. If the client specified some NLS settings, then the remaining NLS settings default.

Each session started on behalf of a client application may run in the same or a different locale as other sessions. For example, one session may use the German locale while another uses the French locale. Also, each session may have the same or different language requirements specified.

[Table 19–1](#) shows two clients using different `NLS_LANG` settings. A user starts SQL*Plus on each host, logs on to the same database as `hr`, and runs the same query

simultaneously. The result for each session differs because of the locale-specific NLS setting for floating-point numbers.

Table 19–1 Locale-Specific NLS Settings

t	Client Host 1	Client Host 2
t0	\$ export NLS_LANG=American_America.US7ASCII	\$ export NLS_LANG=German_Germany.US7ASCII
t1	<pre>\$ sqlplus /nolog SQL> CONNECT hr@proddb Enter password: ***** SQL> SELECT 999/10 FROM DUAL;</pre> <p>999/10 ----- 99.9</p>	<pre>\$ sqlplus /nolog SQL> CONNECT hr@proddb Enter password: ***** SQL> SELECT 999/10 FROM DUAL;</pre> <p>999/10 ----- 99,9</p>

See Also: *Oracle Database 2 Day + Java Developer's Guide* and *Oracle Database Globalization Support Guide* to learn about NLS settings

Oracle Globalization Development Kit

The **Oracle Globalization Development Kit (GDK)** simplifies the development process and reduces the cost of developing Internet applications used to support a global environment. The GDK 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.

The GDK mainly consists of two parts: GDK for Java and GDK for PL/SQL. GDK for Java provides globalization support to Java applications. GDK for PL/SQL provides globalization support to the PL/SQL programming environment. The features offered in the two parts are not identical.

See Also: *Oracle Database Globalization Support Guide*

Unstructured Data

The traditional relational model deals with simple structured data that fits into simple tables. Oracle Database also provides support for **unstructured data**, which cannot be decomposed into standard components. Unstructured data includes text, graphic images, video clips, and sound waveforms.

Oracle Database includes data types to handle unstructured content. These data types appear as native types in the database and can be queried using SQL.

Overview of XML in Oracle Database

Oracle XML DB is a set of Oracle Database technologies related to high-performance XML storage and retrieval. XML DB provides native XML support by encompassing both SQL and XML data models in an interoperable manner.

Oracle XML DB is suited for any Java or PL/SQL application where some or all of the data processed by the application is represented using XML. For example, the application may have large numbers of XML documents that must be ingested, generated, validated, and searched.

Oracle XML DB provides many features, including the following:

- The native `XMLType` data type, which can represent an XML document in the database so that it is accessible by SQL

- Support for XML standards such as XML Schema, XPath, XQuery, XSLT, and DOM
- XMLIndex, which supports all forms of XML data, from highly structured to completely unstructured

Example 19–1 creates a table `orders` of type `XMLType`. The example also creates a SQL **directory object**, which is a logical name in the database for a physical directory on the host computer. This directory contains XML files. The example inserts XML content from the `purchaseOrder.xml` file into the `orders` table.

Example 19–1 XMLType

```
CREATE TABLE orders OF XMLType;  
CREATE DIRECTORY xmldir AS path_to_folder_containing_XML_file;  
INSERT INTO orders VALUES (XMLType(bfilename('XMLDIR', 'purchaseOrder.xml'),  
                                nls_charset_id('AL32UTF8')));
```

The **Oracle XML developer's kits (XDK)** contain 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 XDKs are fully supported and come with a commercial redistribution license.

See Also:

- *Oracle XML DB Developer's Guide*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database XML Java API Reference*

Overview of LOBs

The **large object (LOB)** data types enable you to store and manipulate large blocks of unstructured data in binary or character format. LOBs provide efficient, random, piece-wise access to the data.

Internal LOBs An **internal LOB** stores data in the database itself rather than in external files. Internal LOBs include the following:

- CLOB (character LOB), which stores large amounts of text, such as text or XML files, in the database character set
- NCLOB (national character set LOB), which stores Unicode data
- BLOB (binary LOB), which stores large amounts of binary information as a bit stream and is not subject to character set translation

The database stores LOBs differently from other data types. Creating a LOB column implicitly creates a **LOB segment** and a **LOB index** (see "[User Segment Creation](#)" on page 12-21). The tablespace containing the LOB segment and LOB index, which are always stored together, may be different from the tablespace containing the table.

Note: Sometimes the database can store small amounts of LOB data in the table itself rather than in a separate LOB segment.

The LOB segment stores data in pieces called **chunks**. A chunk is a logically contiguous set of data blocks and is the smallest unit of allocation for a LOB. A row in the table stores a pointer called a **LOB locator**, which points to the LOB index. When the table is queried, the database uses the LOB index to quickly locate the LOB chunks.

The database manages **read consistency** for LOB segments differently from other data (see "[Read Consistency and Undo Segments](#)" on page 9-3). Instead of using **undo data** to record changes, the database stores the before images in the segment itself. When a transaction changes a LOB, the database allocates a new chunk and leaves the old data in place. If the transaction rolls back, then the database rolls back the changes to the index, which points to the old chunk.

External LOBs A `BFILE` (binary file LOB) is an **external LOB** because the database stores a pointer to a file in the operating system. The external data is read-only.

A `BFILE` uses a directory object to locate data. The amount of space consumed depends on the length of the directory object name and the length of the file name.

A `BFILE` does not use the same read consistency mechanism as internal LOBs because the binary file is external to the database. If the data in the file changes, then repeated reads from the same binary file may produce different results.

SecureFiles SecureFiles is a LOB data type specifically engineered to deliver high performance for file data comparable to that of traditional file systems, while retaining the advantages of Oracle Database. The `SECUREFILE` LOB parameter enables advanced features typically found in high-end file systems, such as deduplication, compression, encryption, and journaling.

See Also:

- "[Oracle Data Types](#)" on page 2-9
- *Oracle Database SecureFiles and Large Objects Developer's Guide* to learn more about LOB data types

Overview of Oracle Text

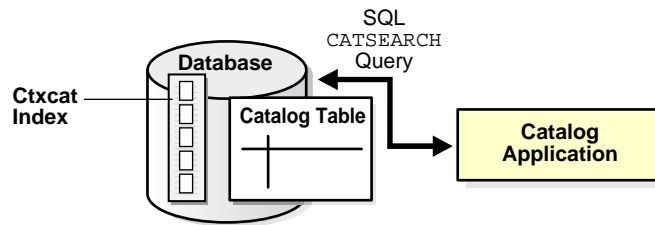
Oracle Text (Text) is a fast and accurate full-text retrieval technology integrated with Oracle Database. Oracle Text indexes any document or textual content stored in file systems, databases, or on the Web. These documents can be searched based on their textual content, metadata, or attributes.

Oracle Text provides the following advantages:

- Oracle Text allows text searches to be combined with regular database searches in a single SQL statement. The Text index is in the database, and Text queries are run in the Oracle Database process. The **optimizer** can choose the best **execution plan** for any query, giving the best performance for ad hoc queries involving Text and structured criteria.
- You can use Oracle Text with XML data. In particular, you can combine `XMLIndex` with Oracle Text indexing, taking advantage of both XML and a full-text index.
- The Oracle Text SQL API makes it simple and intuitive to create and maintain Oracle Text indexes and run searches.

For a use case, suppose you must create a catalog index for an auction site that sells electronic equipment. New inventory is added every day. Item descriptions, bid dates, and prices must be stored together. The application requires good response time for mixed queries. First, you create and populate a `catalog` table. You then use Oracle Text to create a `CTXCAT` index that you can query with the `CATSEARCH` operator in a `SELECT ... WHERE CATSEARCH` statement.

[Figure 19-4](#) illustrates the relation of the catalog table, its `CTXCAT` index, and the catalog application that uses the `CATSEARCH` operator to query the index.

Figure 19–4 Catalog Query Application**See Also:**

- *Oracle Text Application Developer's Guide* and *Oracle Text Reference*
- *Oracle XML DB Developer's Guide* to learn how to perform full-text search over XML data

Overview of Oracle Multimedia

Oracle Multimedia enables Oracle Database to store, manage, and retrieve images, medical images that follow the Digital Imaging and Communications in Medicine (DICOM) standard, audio, and video data in an integrated fashion with other enterprise information. Oracle Multimedia provides object types and methods for:

- Extracting metadata and attributes from multimedia data
- Embedding metadata created by applications into image and DICOM data
- Obtaining and managing multimedia data from Oracle Database, Web servers, file systems, and other servers
- Performing operations such as thumbnail generation on image and DICOM data
- Making DICOM data anonymous
- Checking DICOM data for conformity to user-defined validation rules

See Also:

- *Oracle Multimedia User's Guide* and *Oracle Multimedia Reference*
- *Oracle Multimedia DICOM Developer's Guide* and *Oracle Multimedia DICOM Java API Reference*
- *Oracle Multimedia Java API Reference* and *Oracle Multimedia Servlets and JSP Java API Reference*

Overview of Oracle Spatial

Oracle Spatial (Spatial) provides a SQL schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial features in an Oracle database. Oracle Spatial makes spatial data management easier to users of location-enabled applications and geographic information system (GIS) applications.

An example of spatial data is a road map. The spatial data indicates the Earth location (such as longitude and latitude) of objects on the map. When the map is rendered, this spatial data is used to project the locations of the objects on a two-dimensional piece of paper. A GIS is often used to store, retrieve, and render this Earth-relative spatial data. When spatial data is stored in an Oracle database, the data can be easily manipulated, retrieved, and related to other data.

See Also: *Oracle Spatial Developer's Guide*

Glossary

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.

active transaction

A transaction that has started but not yet committed or rolled back.

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 diagnostic information, including network tracing and logging.

alert log

A file that provides a chronological log of database messages and errors. The alert log is stored in the [ADR](#).

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](#).

Automatic Database Diagnostic Monitor (ADDM)

See [ADDM](#).

Automatic Diagnostic Repository (ADR)

See [ADR](#).

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](#).

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.

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 [instance](#); [Oracle process](#).

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 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 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.

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.

cache recovery

The 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.

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 (DBWn)** process writes blocks to disk to synchronize the buffer cache with the **data files**.

client

In **client/server architecture**, the front-end database application that interacts with a user. The client portion has no data access responsibilities.

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.

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.

commit

Action that ends a database **transaction** and makes permanent all changes performed in the transaction.

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**.

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.

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. In other words, the backup does not require the application of redo to be made consistent.

See also **inconsistent backup**.

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.

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.

database

Organized collection of data treated as a unit. The purpose of a database is to store and retrieve related information. Every Oracle database **instance** accesses only one database in its lifetime.

database buffer cache

The portion of the **system global area (SGA)** that holds copies of **data blocks**. All **client processes** concurrently connected to the **instance** share access to the buffer cache.

database link

In a **schema object**, a **schema object** in one database that enables users to access objects on a different database.

database user

An account through which you can log in to an Oracle database.

database writer (DBWn)

A **background process** that writes buffers in the **database buffer cache** to **data files**.

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 consistency

A consistent view of the data by each user in a multiuser database.

See also [concurrency](#).

data dictionary

A read-only collection of database tables and views containing reference information about the database, its structures, and its users.

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 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](#).

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.

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 after the transaction is committed. A deferrable constraint enables you to disable the constraint temporarily while making changes that might violate the constraint.

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.

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.

dispatcher process (Dnnn)

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 processing

The operations that occurs 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`.

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 down time.

ETL

Extraction, transformation, and loading (ETL). The process of extracting data from source systems and bringing it into a [data warehouse](#).

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 [hints](#).

expression

A combination of one or more values, [operators](#), and SQL functions that evaluates 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 is 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 the index blocks in no particular order. The database accesses the data in the index itself, without accessing the table.

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.

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.

full index scan

A scan of an [index](#) in which the database reads the entire index in order.

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. All [data blocks](#) under the [high water mark](#) are scanned.

function

A schema object, similar to a [procedure](#), that always returns a single value.

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 Restart and Oracle ASM. Oracle Database 11g Release 2 (11.2) combines these infrastructure products into one software installation called the [grid infrastructure 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 function

A [function](#) that operates on an arbitrary-length input value and returns a fixed-length hash value.

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.

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 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.

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.

high water mark

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.

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.

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.

index

Optional schema object associated with a nonclustered **table**, table **partition**, or **table cluster**. In some cases indexes speed data access.

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-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.

index segment

A **segment** that stores data for a nonpartitioned index or index **partition**.

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**.

instance

The combination of the **system global area (SGA)** and **background processes**. An instance is associated with one and only one database. In an Oracle Real Application Clusters configuration, multiple instances access a single database simultaneously.

instance failure

The termination of a database **instance** because of a hardware failure, Oracle internal error, or SHUTDOWN ABORT statement.

instance recovery

The automatic application of redo log records to uncommitted data blocks when an **instance** is restarted after a failure.

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.

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 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`.

key

Column or set of columns included in the definition of certain types of **integrity constraints**.

key compression

The elimination of repeated occurrence of **primary key** column values in an **index-organized table**.

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.

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.

listener

A process that listens for incoming client connection requests and manages network traffic to the database.

LOB

Large object. An Oracle data type designed to hold large amounts of data.

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.

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 (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 (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.

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.

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-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.

master site

In a **replication** environment, a different database with which a **materialized view** shares data.

master table

The table associated with a **materialized view** at a **master site**.

materialized view

A schema object that stores the result of a query. Oracle materialized views can be read-only or updatable.

See also **view**.

media recovery

The application of redo or incremental backups to a **data block** or backup **data file**.

mounted database

An **instance** that is started and has the database **control file** open.

null

Absence of a value in a **column** of a **row**. Nulls indicate missing, unknown, or inapplicable data.

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.

OLAP

Online Analytical Processing. OLAP is characterized by dynamic, dimensional analysis of historical data.

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. **log writer (LGWR)** writes the contents of the redo buffer to the online redo log.

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. The SQL operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*).

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.

Oracle architecture

Memory and **process** structures used by Oracle Database to manage a **database**.

Oracle Automatic Storage Management (Oracle ASM)

See **Oracle ASM**.

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, file systems, and raw devices.

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 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 Enterprise Manager

A system management tool that provides centralized management of an Oracle database environment.

Oracle home

The operating system location of an Oracle Database installation.

Oracle process

A [process](#) that runs Oracle Database code. 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 XA

An external interface that allows global transactions to be coordinated by a [transaction manager](#) other than Oracle Database.

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.

PGA

Program global area. A memory buffer that contains data and control information for a [server process](#).

See also [SGA](#).

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.

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.

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.

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.

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.

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**.

program global area (PGA)

See **PGA**.

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 plan

The **execution plan** used to execute a query.

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 **concurrency**; **data consistency**.

read-only database

A database that is available for queries only and cannot be modified.

Recovery Manager (RMAN)

See [RMAN](#).

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 is used if [instance recovery](#) is necessary.

redo thread

The redo generated by a database [instance](#).

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).

replication

The process of sharing database objects and data at multiple databases.

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.

rowid

A globally unique address for a row in a database.

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.

savepoint

A named SCN in a transaction to which the transaction can be rolled back.

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.

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.

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.

selectivity

In a query, the measure of how many rows from a row set pass a [predicate](#) test, for example, `WHERE last_name = 'Smith'`. A selectivity of 0.0 means no rows, whereas a value of 1.0 means all rows. A predicate becomes more selective as the value approaches 0.0 and less selective (or more unselective) as the value approaches 1.0.

sequence

A [schema object](#) that generates a serial list of unique numbers for table columns.

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.

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](#).

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.

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](#).

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*Plus

Oracle tool used to run [SQL](#) statements against Oracle Database.

standby database

An independent copy of a production database that you can use for disaster protection in a high availability environment.

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.

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.

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 global area (SGA)

See [SGA](#).

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](#).

tablespace

A database storage unit that groups related logical structures together. The database [data files](#) are stored in tablespaces.

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.

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 ID

An identifier is unique to a [transaction](#) and represents the undo segment number, slot, and sequence number.

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.

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.

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 tablespace

A [tablespace](#) containing undo segments when [automatic undo management mode](#) is enabled.

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 process

See [client process](#).

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.

whole database backup

A backup of the [control file](#) and all [data files](#) that belong to a database.

Index

A

access drivers, external table, 2-17
access paths, data, 3-2, 3-19, 7-10, 7-12, 7-21
accounts, user, 6-1
ACID properties, 10-1
active transactions, 10-7
ADDM (Automatic Database Diagnostic Monitor), 18-22, 18-23
administrative accounts, 2-5, 6-2
administrator privileges, 2-5, 13-6, 16-14
Advanced Queuing, Oracle Streams, 15-11, 17-23
alert logs, 13-21, 15-10
ALL_ data dictionary views, 6-3
ALTER SESSION statement, 7-9
anonymous PL/SQL blocks, 8-2
ANSI/ISO standard, 7-2
APIs (application program interfaces), 16-16
 embedded SQL statements, 7-9
 external tables, 2-17
 Java, 8-13, 17-20, 19-6, 19-8
 JDBC, 7-9, 8-15
 network services, 16-6
 OCI/OCCL, 19-7
 ODBC, 7-9, 19-8
 Oracle Data Pump, 18-7
 Oracle Streams Advanced Queuing, 17-24
 SQLJ, 8-15
application architecture, 1-10
application developers
 duties of, 19-1
 tools for, 19-1
 topics for, 19-3
application domain indexes, 3-19
application processes, 15-3
application program interface. *See* API
application servers, 1-11, 8-14
archived redo log files, 11-11, 11-15, 18-14
ARCHIVELOG mode, 15-12
archiver process (ARCn), 15-12
ascending indexes, 3-11
ASM (Automatic Storage Management), 11-3, 11-4, 17-14
atomicity, statement-level, 10-4
AUDIT statement, 7-3
auditing, 6-1, 6-5, 7-3, 8-17, 11-6, 13-8, 16-4, 17-5

Oracle Audit Vault, 17-5
authentication, database, 7-9, 15-4, 17-3
Automatic Database Diagnostic Monitor. *See* ADDM
automatic maintenance tasks, 18-22
automatic memory management, 18-15, 18-16
Automatic Storage Management. *See* ASM
automatic undo management, 12-24
AutoTask, 18-22

B

background processes, 15-7
backups, database, 18-11
bitmap indexes, 3-13, 4-9
 bitmap joins, 3-15
 locks, 3-14
 mapping table, 3-25
 single-table, 3-14
 storage, 3-17
bitmap tablespace management, 12-3
blocking transactions, 9-7
blocks, data. *See* data blocks
BOOLEAN data type, 2-9, 3-7, 7-5, 8-19
branch blocks, index, 3-5
B-tree indexes, 2-19, 2-23, 3-5
 branch level, 3-6
 height, 3-6
 key compression, 3-12
 reverse key, 3-11
buffer cache, database. *See* database buffer cache
buffers. *See* database buffers
business rules, enforcing, 5-1

C

cache fusion, 9-3
cardinality, column, 3-13, 7-12
Cartesian joins, 7-7
cartridges, 3-19
cascading deletions, 5-8
catalog.sql script, 6-6
chaining, rows. *See* row chaining
character data types, 2-10
 byte semantics, 2-10
 CHAR, 2-10
 character semantics, 2-10

- VARCHAR2, 2-10
- character sets, 2-10
 - ASCII, 2-10
 - EBCDIC, 2-10
 - Unicode, 2-10
- check constraints, 5-3, 5-9
- checkpoint process (CKPT), 15-10
- checkpoints
 - control files, 11-10
 - database shutdowns, 13-9
 - definition, 13-11
 - inconsistent backups, 18-11
 - incremental, 13-11
 - position, 13-14
 - thread, 13-11
- client processes, 15-3
 - connections and, 15-4
 - sessions and, 15-4
 - shared server processes and, 16-14
- client result cache, 14-20
- client/server architecture, 16-1
- client-side programming, 8-1
- cluster indexes, 2-23
- clusters, table, 2-25
 - cluster keys, 2-22
 - hash, 2-25
 - index, 2-25
- Codd, E. F., 1-2
- code points, 2-10
- collections, PL/SQL, 8-10
- columns
 - cardinality, 3-13, 7-12
 - definition, 2-7
 - order of, 2-18
 - prohibiting nulls in, 5-3
 - virtual, 2-7, 2-18, 3-19
- COMMENT statement, 7-3
- committing transactions
 - COMMIT statement, 7-8
 - defined, 10-1
 - ending transactions, 10-3
 - fast commit, 15-9
 - group commits, 15-10
 - implementation, 15-9
 - implicit commits, 7-4
 - two-phase commit, 10-13
- compiled PL/SQL
 - pseudocode, 8-21
 - shared pool, 8-11
 - triggers, 8-21
- complete recovery, 18-14
- composite indexes, 3-3
- composite partitioning, 4-2
- compound triggers, 8-18
- compression
 - basic table, 2-19
 - data block, 12-11
 - Hybrid Columnar Compression, 2-20
 - index key, 3-12
 - OLTP table, 2-20
 - online archival, 2-21
 - table, 2-19, 4-7
 - warehouse, 2-21
- concatenated indexes, 3-3
- concurrency
 - definition, 9-1
 - dirty reads, 9-5
 - fuzzy reads, 9-5
 - phantom reads, 9-5
 - row locks, 9-19
 - transaction isolation, 9-5, 9-8, 9-11
- conditions, SQL, 7-2, 7-5
- conflicting writes, 9-7
- connections, client/server
 - administrator privileges, 13-6
 - defined, 15-4
 - embedded SQL, 7-9
 - listener process, 16-6
 - sessions contrasted with, 15-4
- consistency
 - conflicting writes, 9-7
 - definition, 9-1
 - multiversioning, 9-1, 9-2
- consistent read clones, 9-4
- constraints, integrity
 - check, 5-3, 5-9
 - default values, 5-12
 - deferrable, 5-5, 5-11, 7-8
 - enabling and disabling, 5-10
 - enforced with indexes, 5-5
 - foreign key, 5-3, 5-6
 - mechanisms of enforcement, 5-12
 - NOT NULL, 2-9, 5-2, 5-3
 - primary key, 2-9, 5-2, 5-5
 - REF, 5-3
 - referential, 5-8
 - self-referential, 5-7
 - state of, 5-10
 - unique key, 5-2, 5-3, 5-4
 - validating, 5-10
- contention
 - for data
 - deadlocks, 9-16
 - lock escalation, 9-15
- contexts, 2-2
- control files, 11-10
 - changes recorded, 11-10
 - checkpoints and, 11-10
 - contents, 11-10
 - multiplexed, 11-11
 - overview, 11-10
 - used in mounting database, 13-7
- CREATE CLUSTER statement, 2-23
- CREATE DIMENSION statement, 4-21, 4-22
- CREATE GLOBAL TEMPORARY TABLE
 - statement, 2-16
- CREATE INDEX statement, 2-16, 3-3, 3-11, 3-12, 3-18, 3-20, 4-11
 - storage parameters, 12-22
- CREATE MATERIALIZED VIEW statement, 4-17

- CREATE SEQUENCE statement, 4-20
- CREATE SYNONYM statement, 4-23
- CREATE TABLE command, 2-7
- CREATE TABLE statement, 2-6
- CREATE TRIGGER statement
 - compiled and stored, 8-21
- CREATE UNIQUE INDEX statement, 5-5
- CREATE USER statement
 - temporary segments, 12-23
- cursors
 - embedded SQL, 7-9
 - explicit, 8-10
 - fetching rows, 7-9

D

- data
 - consistency of
 - locks, 9-5
 - manual locking, 9-26
 - data blocks, 7-22, 12-1
 - cached in memory, 14-13
 - clustered rows, 2-22
 - coalescing free space in blocks, 12-13
 - compression, 12-11
 - format, 12-7, 12-8
 - locks stored in, 9-20
 - overview, 12-2
 - shown in rowids, 12-10
 - stored in the buffer cache, 14-9
 - writing to disk, 14-13
 - data conversion
 - program interface, 16-16
 - data dictionary, 2-5, 2-15
 - ALL_ prefixed views, 6-3
 - cache, 14-15
 - comments in, 7-3
 - content, 6-2, 14-19
 - DBA_ prefixed views, 6-3
 - dictionary managed tablespaces, 12-6
 - DUAL table, 6-4
 - dynamic performance views, 6-5
 - locks, 9-24
 - overview, 6-1
 - owner, 6-4
 - public synonyms, 6-5
 - row cache and, 14-19
 - stored subprograms, 8-4
 - USER_ prefixed views, 6-4
 - uses, 6-5
 - data dictionary cache, 6-5, 7-17, 14-15, 14-19
 - data failures, protecting against human errors, 17-9
 - data files
 - contents of, 11-9
 - data file 1, 12-32
 - named in control files, 11-10
 - shown in rowids, 12-10
 - SYSTEM tablespace, 12-32
 - temporary, 11-8
 - data integrity, 5-1
 - enforcing, 5-1, 6-4
 - SQL and, 7-1
 - data manipulation language. *See* DML
 - data object number
 - extended rowid, 12-10
 - Data Pump Export
 - dump file set, 18-7
 - Data Recovery Advisor, 18-13
 - data segments, 12-21
 - data types
 - BOOLEAN, 2-9, 3-7, 7-5, 8-19
 - built-in, 2-9
 - CHAR, 2-10
 - character, 2-10
 - composite types, 2-9
 - conversions of
 - by program interface, 16-16
 - DATE, 2-12
 - datetime, 2-12
 - definition, 2-9
 - format models, 2-12
 - how they relate to tables, 2-7
 - in PL/SQL, 2-9
 - LONG
 - storage of, 2-18
 - NCHAR, 2-11
 - NUMBER, 2-11
 - numeric, 2-11
 - NVARCHAR2, 2-11
 - object, 2-15
 - reference types, 2-9
 - ROWID, 2-13
 - TIMESTAMP, 2-12
 - UROWID, 2-13
 - user-defined, 2-9, 4-16
 - data warehouses
 - architecture, 17-16
 - bitmap indexes in, 3-13
 - dimensions, 4-21
 - materialized views, 4-16
 - partitioning in, 4-2
 - summaries, 4-16
 - database applications
 - definition, 1-1
 - database authentication, 7-9, 15-4
 - database backups, 18-11
 - database buffer cache, 2-19, 14-9, 15-8
 - cache hits and misses, 14-11
 - caching of comments, 6-5
 - flash cache, 14-9
 - database buffers
 - after committing transactions, 10-11
 - buffer bodies in flash cache, 14-11
 - buffer cache, 14-9
 - checkpoint position, 15-9
 - committing transactions, 15-9
 - defined, 14-9
 - writing, 15-8
 - Database Configuration Assistant, 6-6
 - database resident connection pooling, 16-14

- Database Server Grid, 17-11
 - description, 17-12
- Database Smart Flash Cache. *See* flash cache
- Database Storage Grid, 17-11
 - description, 17-14
- database structures
 - control files, 11-10
 - data blocks, 12-1, 12-6
 - data files, 11-1
 - extents, 12-1
 - processes, 15-1
 - segments, 12-1, 12-21
 - tablespaces, 11-1, 12-30
- database writer process (DBW*n*), 15-8
 - checkpoints, 15-9
 - defined, 15-8
 - least recently used algorithm (LRU), 14-13
 - multiple DBW*n* processes, 15-8
 - write-ahead, 15-9
- databases
 - administrative accounts, 2-5
 - character sets, 2-10
 - closing, 13-10
 - terminating the instance, 13-10
 - definition, 1-1
 - distributed
 - changing global database name, 14-19
 - hierarchical, 1-2
 - history, 1-3
 - incarnations, 18-14
 - introduction, 1-1
 - mounting, 13-7
 - name stored in control files, 11-10
 - network, 1-2
 - object-relational, 1-3
 - opening, 13-7
 - relational, 1-2, 7-1
 - shutting down, 13-8
 - starting up, 2-5, 13-1
 - forced, 13-10
 - structures
 - control files, 11-10
 - data blocks, 12-1, 12-6
 - data files, 11-1
 - extents, 12-1, 12-18
 - logical, 12-1
 - processes, 15-1
 - segments, 12-1, 12-21
 - tablespaces, 11-1, 12-30
- DATE data type, 2-12
- datetime data types, 2-12
- DBA_ views, 6-3
- DBMS (database management system), 1-1
- DBMS_METADATA package, 6-6
- DBMS_STATS package, 7-14
- DBW*n* background process, 15-8
- DDL (data definition language), 6-1
 - described, 7-3
 - locks, 9-24
 - processing of, 7-23
- deadlocks, 7-17
 - defined, 9-16
- decision support systems (DSS)
 - materialized views, 4-16
- dedicated server, 7-18
- default values
 - constraints effect on, 5-12
- definer's rights, 8-4
- degree of parallelism
 - parallel SQL, 15-15
- DELETE statement, 7-4
 - freeing space in data blocks, 12-13
- deletions, cascading, 5-8
- denormalized tables, 4-22
- dependencies, schema object, 2-4
- descending indexes, 3-11
- detail tables, 4-16
- dictionary cache locks, 9-26
- dictionary managed tablespaces, 12-6
- dimensions, 4-21
 - attributes, 4-22
 - hierarchies, 4-22
 - join key, 4-22
 - normalized or denormalized tables, 4-22
 - tables, 4-21
- directory objects, 2-2
- dirty reads, 9-2, 9-5
- disk space
 - data files used to allocate, 11-9
- dispatcher processes
 - described, 16-13
- dispatcher processes (D*nnn*)
 - client processes connect through Oracle Net Services, 16-11, 16-13
 - network protocols and, 16-13
 - prevent startup and shutdown, 16-14
 - response queue and, 16-12
- distributed databases
 - client/server architectures and, 16-2
 - job queue processes, 15-12
 - recoverer process (RECO) and, 15-11
 - server can also be client in, 16-2
 - transactions, 10-12
- distributed transactions, 10-7, 10-12
 - in-doubt, 10-13
 - naming, 10-7
 - two-phase commit and, 10-13
- DML (data manipulation language)
 - indexed columns, 3-14
 - invisible indexes, 3-2
 - locks, 9-18
 - overview, 7-4
 - referential actions, 5-9
 - triggers, 8-17
- downtime
 - avoiding during planned maintenance, 17-9
 - avoiding during unplanned maintenance, 17-7
- drivers, 16-16
- DUAL table, 6-4
- dynamic partitioning, 15-15

dynamic performance views (V\$ tables), 6-5
dynamic SQL
 DBMS_SQL package, 8-10
 embedded, 8-10

E

embedded SQL, 7-1, 7-9, 8-16
enqueued transactions, 10-9
Enterprise Grids
 with Oracle Real Application Clusters, 17-11
Enterprise Manager
 alert log, 13-21
 dynamic performance views usage, 6-5
 executing a package, 8-8
 lock and latch monitors, 9-25
 shutdown, 13-9, 13-10
 SQL statements, 7-2
equijoins, 3-17
exceptions, PL/SQL, 8-10
exclusive locks, 9-15
 row locks (TX), 9-18
 table locks (TM), 9-20
EXECUTE statement, 8-6
execution plans, 4-19, 7-10, 7-12
 EXPLAIN PLAN, 7-5
EXPLAIN PLAN statement, 7-5, 7-12
explicit locking, 9-26
expressions, SQL, 3-3, 7-5
extents
 as collections of data blocks, 12-18
 defined, 12-2
 dictionary managed, 12-6
 incremental, 12-19
 locally managed, 12-3
 overview of, 12-18
external procedures, 8-11
external tables, 2-7
extraction, transformation, and loading (ETL)
 overview, 17-18

F

fact tables, 4-21
failures
 database buffers and, 13-12
 statement and process, 15-8
fast commit, 15-9
fast full index scans, 3-7
fast refresh, 4-18
fast-start
 rollback on demand, 13-14
fields, 2-9
file management locks, 9-26
files
 alert log, 15-10
 initialization parameter, 13-6, 13-15
 password
 administrator privileges, 13-6
 server parameter, 13-6, 13-15

 trace files, 15-10
fixed views, 6-6
flash cache
 buffer reads, 14-11
 definition, 14-9
 optimized physical reads, 14-11
floating-point numbers, 2-12
foreign key constraints, 5-3
foreign keys, 2-9, 5-6
 changes in parent key values, 5-8
 composite, 5-6
 indexing, 3-3
 updating parent key tables, 5-8
 updating tables, 9-21
format models, data type, 2-12, 2-14
free space
 automatic segment space management, 12-11
 managing, 12-11
full index scans, 3-6
full table scans, 3-1, 3-7, 5-9, 7-12
 LRU algorithm and, 14-13
 parallel exe, 15-15
function-based indexes, 3-17
functions, 7-5
 function-based indexes, 3-17
 hash, 4-5
 PL/SQL, 8-3
 SQL, 2-13
fuzzy reads, 9-5

G

global database names
 shared pool and, 14-19
global indexes, 4-7, 4-10
GRANT statement, 4-23, 7-3
grid computing
 Database Server Grid, 17-11
 Database Storage Grid, 17-11
group commits, 15-10
GV\$ views, 6-6

H

handles for SQL statements, 14-5
hard parsing, 7-17
hash clusters, 2-25
 cluster keys, 2-25
 hash collisions, 2-27
 hash key values, 2-25
 queries, 2-26
 single-table, 2-26
 storage, 2-27
hash functions, 2-25, 4-5
hash partitions, 4-5
headers, data block, 9-20
headers, data blocks, 12-8
Health Monitor, 18-13
heap-organized tables, 2-3, 3-20
height, index, 3-6

- hierarchies
 - join key, 4-22
 - levels, 4-22
- hints, optimizer, 7-10, 7-14
- Hybrid Columnar Compression, 2-20

I

- implicit queries, 7-7
- incarnations, database, 18-14
- incremental refresh, 4-18
- index unique scans, 3-8
- indexes
 - application domain, 3-19
 - ascending, 3-11
 - bitmap, 3-13, 4-9
 - bitmap join, 3-15
 - branch blocks, 3-5
 - B-tree, 2-19, 3-5
 - cardinality, 3-13
 - composite, 3-3
 - compressed, 3-13
 - concatenated, 3-3
 - descending, 3-11
 - domain, 3-19
 - enforcing integrity constraints, 5-5
 - extensible, 3-19
 - function-based, 3-17
 - global, 4-7, 4-10
 - invisible, 3-2
 - keys, 3-3, 3-12, 5-5
 - leaf blocks, 3-5
 - nonprefixed, local, 4-9
 - nonunique, 3-4
 - overview, 3-1
 - partitions, 4-7
 - prefixed, local, 4-9
 - reverse key, 3-11
 - scans, 3-6, 3-7, 3-8, 7-13
 - secondary, 3-23
 - segments, 3-17
 - selectivity, 3-7
 - storage, 3-20
 - types, 3-4
 - unique, 3-4
- indexes, local, 4-7
- index-organized tables, 3-20, 3-24
 - benefits, 3-20
 - partitioned, 4-12
 - row overflow, 3-23
 - secondary indexes, 3-23
- information systems, 1-1
- initialization parameter file, 13-6, 13-15
 - startup, 13-6
- initialization parameters
 - basic, 13-15
 - OPEN_CURSORS, 14-6
 - SERVICE_NAMES, 16-9
- INIT.ORA. *See* initialization parameter file.
- inner joins, 7-7

- INSERT statement, 7-4
- instance PGA
 - definition, 14-2
- instance recovery
 - SMON process, 15-8
- instances, 7-9
 - associating with databases, 13-7
 - described, 13-1
 - failures, 9-18
 - memory structures of, 14-1
 - multiple-process, 15-1
 - process structure, 15-1
 - recovery of
 - SMON process, 15-8
 - service names, 16-6
 - shutting down, 13-8, 13-10
 - terminating, 13-10
- INSTEAD OF triggers, 8-18
- integrity constraints, 5-1
 - advantages, 5-1
 - check, 5-9
 - definition, 2-14
 - inline, 5-2
 - out-of-line, 5-2
 - views, 4-14
- interested transaction lists (ITLs), 9-5
- internal locks, 9-26
- invisible indexes, 3-2
- invoker's rights, 8-4
- isolation levels
 - read-only, 9-11
 - serialization, 9-8
 - setting, 9-26
- isolation levels, transaction, 9-5
 - read committed, 9-6

J

- Java
 - call specifications, 8-15
 - overview, 8-12
 - SQLJ translator, 8-16
 - stored procedures, 1-5, 8-15
 - virtual machine, 8-13
- JDBC
 - accessing SQL, 8-15
 - driver types, 8-15
 - drivers, 8-15
 - embedded SQL, 7-9
- job queue processes, 15-12
- jobs, 15-1
- join views, 4-15
- joins, 6-2
 - nested loop, 7-21
 - views, 4-15
- joins, table, 3-15, 7-6
 - Cartesian, 7-7
 - clustered tables, 2-23
 - conditions, 3-17
 - inner joins, 7-7

- join conditions, 7-6
- outer joins, 7-7
- views, 4-12

K

- key-preserved tables, 4-15

- keys

- compression, 3-12
- concatenation of index, 3-12
- foreign, 5-6
- indexes, 3-3, 3-12, 5-5
- natural, 5-5
- parent, 5-6
- partition, 4-2
- prefixed index, 3-6, 3-12
- referenced, 5-6
- reverse, 3-11
- surrogate, 5-5
- unique, 5-3
- values, 5-2

L

- large pool, 14-21

- latches

- definition, 9-25
- enqueue, 9-25
- parsing and, 7-17
- sleeping, 9-25
- spinning, 9-25

- leaf blocks, index, 3-5

- least recently used (LRU) algorithm

- database buffers and, 14-9
- full table scans and, 14-13
- latches, 14-13
- shared SQL pool, 14-19

- LGWR background process, 15-9

- library cache, 7-17, 14-15, 14-16

- list partitions, 4-4

- listener process, 16-6

- service names, 16-6

- listeners, 16-6

- service names, 16-6

- local indexes, 4-7

- locally managed tablespaces, 12-3

- LOCK TABLE statement, 7-5

- locks, 9-5

- after committing transactions, 10-10
- automatic, 9-13, 9-17
- bitmap indexes, 3-14
- conversion, 9-15
- deadlocks, 7-17, 9-16
- dictionary, 9-24
- dictionary cache, 9-26
- DML, 9-18
- duration, 9-13, 9-16
- escalation, 9-15
- exclusive, 9-15
- exclusive DDL, 9-24

- exclusive table, 9-21
- file management locks, 9-26
- latches, 9-25

- log management locks, 9-26

- manual, 9-26

- overview of, 9-5

- parse, 9-24

- restrictiveness, 9-15

- rollback segments, 9-26

- row (TX), 9-18

- row exclusive table, 9-21

- row share, 9-21

- share DDL, 9-24

- share locks, 9-15

- share row exclusive, 9-21

- share table, 9-21

- system, 9-25

- table, 3-1, 7-5

- table (TM), 9-20

- tablespace, 9-26

- types of, 9-17

- unindexed foreign keys and, 9-21

- user-defined, 9-27

- log management locks, 9-26

- log switch

- archiver process, 15-12

- log switches

- log sequence numbers, 11-13

- log writer process (LGWR), 15-9

- group commits, 15-10

- online redo logs available for use, 11-13

- redo log buffers and, 14-14

- write-ahead, 15-9

- writing to online redo log files, 11-13

- logical database structures

- definition, 1-9

- tablespaces, 12-30

- logical rowids, 3-24

- LONG data type

- storage of, 2-18

- lost updates, 9-7

- LRU, 14-9, 14-13

- shared SQL pool, 14-19

M

- maintenance tasks, automatic, 18-22

- maintenance window, 18-22

- manual locking, 9-26

- mapping tables, 3-25

- master tables, 4-16, 4-17

- materialized views, 4-16

- log, 4-18

- partitioned, 4-18

- refresh

- job queue processes, 15-12

- refreshing, 4-18

- media recovery

- complete, 18-14

- overview, 18-14

- memory
 - allocation for SQL statements, 14-17
 - content of, 14-1
 - processes use of, 15-1
 - software code areas, 14-23
 - stored procedures, 8-3
- memory management
 - about, 18-15
 - automatic, 18-15
 - automatic shared, 18-16
- MERGE statement, 7-4
- metrics, 6-5, 18-21
- monitoring user actions, 17-5
- multiblock writes, 15-9
- multiple-process systems (multiuser systems), 15-1
- multiplexing
 - control files, 11-11
 - redo log file groups, 11-14
 - redo log files, 11-14
- multiuser environments, 15-1
- multiversion read consistency, 6-6, 7-22, 9-1, 9-2, 9-5
 - consistent read clones, 9-4
 - dirty reads, 9-2
 - read committed isolation level, 9-7
 - statement-level, 1-7, 9-2
 - transaction-level, 9-2
 - undo segments, 9-3
- mutexes, 9-25

N

- natural keys, 5-5
- NCHAR data type, 2-11
- network listener process
 - connection requests, 16-13
- networks
 - client/server architecture use of, 16-1
 - communication protocols, 16-16
 - dispatcher processes and, 16-13
 - drivers, 16-16
 - listener processes of, 16-6
 - Oracle Net Services, 16-5
- NLS_DATE_FORMAT parameter, 2-12
- NOAUDIT statement, 7-3
- nonunique indexes, 3-4
- normalized tables, 4-22
- NOT NULL constraints, 5-2, 5-3
- nulls, 2-9
 - foreign keys, 5-8
 - how stored, 2-9, 2-19
 - indexed, 3-4
 - prohibiting, 5-3
- NUMBER data type, 2-11
- numeric data types, 2-11
 - floating-point numbers, 2-12
- NVARCHAR2 data type, 2-11

O

- object tables, 2-15

- object types, 2-15, 4-16
- object views, 4-16
- OLAP
 - index-organized tables, 3-20
 - introduction, 17-19
- OLTP
 - table compression, 2-20
- online analytical processing
 - See* OLAP
- online redo logs
 - checkpoints, 11-10
- OPEN_CURSORS parameter
 - managing private SQL areas, 14-6
- operating systems
 - communications software, 16-16
 - privileges for administrator, 13-6
- operators, SQL, 7-5
- optimization, SQL, 7-19
- optimized physical reads, 14-11
- optimizer, 7-2, 7-10
 - components, 7-11
 - estimator, 7-12
 - execution, 7-20
 - execution plans, 4-19, 7-10, 7-19
 - function-based indexes, 3-19
 - hints, 7-10, 7-14
 - invisible indexes, 3-2
 - partitions in query plans, 4-1
 - plan generator, 7-12
 - query plans, 7-19
 - query rewrite, 4-19
 - query transformer, 4-19, 7-11
 - row sources, 7-19
 - statistics, 2-18, 7-13
 - statistics gathering, 18-22
- Oracle
 - configurations of
 - multiple-process Oracle, 15-1
 - instances, 13-1
 - processes of, 15-6
- Oracle Audit Vault, 17-5
- Oracle blocks, 12-2
- Oracle Call Interface *See* OCI
- Oracle code, 16-16
- Oracle Enterprise Manager. *See* Enterprise Manager
- Oracle Flashback Technology, 18-13
- Oracle *interMedia*
 - See* Oracle Multimedia
- Oracle Internet Directory, 16-9
- Oracle JVM
 - main components, 8-14
 - overview, 8-13
- Oracle Multimedia, 19-14
- Oracle Net Services, 16-5
 - client/server systems use of, 16-5
 - overview, 16-5
 - shared server requirement, 16-13
- Oracle Net Services Connection Manager, 8-12
- Oracle program interface (OPI), 16-16
- Oracle RAC. *See* Oracle Real Application Clusters

- Oracle Real Application Clusters, 6-6
 - Enterprise Grids, 17-11
 - reverse key indexes, 3-11
- Oracle Streams, 17-21
- Oracle Text, 19-13
- Oracle XA
 - session memory in the large pool, 14-21
- ORDBMS (object-relational database management system), 1-3
- outer joins, 7-7

P

- packages, 8-6
 - advantages of, 8-6
 - executing, 8-11
 - for locking, 9-27
 - private, 8-7
 - public, 8-7
 - shared SQL areas and, 14-18
- pages, 12-2
- parallel execution, 15-14
 - coordinator, 15-15
 - server, 15-15
 - servers, 15-15
 - tuning, 15-14
- parallel execution processing, 14
- parallel SQL, 15-14
 - coordinator process, 15-15
 - server processes, 15-15
- parameter
 - server, 13-15
- parameters
 - initialization, 13-15
 - locking behavior, 9-17
 - storage, 12-20
- parse locks, 9-24
- parsing
 - embedded SQL, 7-9
 - storage of information, 6-5
- parsing, SQL, 7-16
 - hard parse, 7-17, 9-25
 - soft parse, 7-17
- partitions, 4-1
 - composite, 4-2
 - dynamic partitioning, 15-15
 - elimination from queries, 4-10
 - hash, 4-5
 - index, 4-7
 - index-organized tables, 4-12
 - keys, 4-2
 - materialized views, 4-18
 - range, 4-2
 - recovery of, 4-8
 - segments, 12-21
 - single-level, 4-2
 - strategies, 4-2
 - table, 4-7
 - tables, 4-12
- passwords
 - administrator privileges, 13-6
 - connecting with, 15-4
- PCTFREE storage parameter
 - how it works, 12-12
- performance
 - dynamic performance views (V\$), 6-5
 - group commits, 15-10
 - packages, 8-7
- PGA, instance
 - definition, 14-2
- phantom reads, 9-5
- physical database structures
 - control files, 11-10
- physical guesses, 3-24
- plan
 - SQL execution, 7-5
- planned downtime
 - avoiding downtime during, 17-9
- PL/SQL
 - anonymous blocks, 8-2, 8-9
 - collections, 8-10
 - data types, 2-9
 - dynamic SQL, 8-10
 - exceptions, 8-10
 - execution, 8-11
 - language constructs, 8-9
 - overview, 8-2
 - packages, 6-6, 8-6
 - PL/SQL engine, 8-11
 - program units, 8-2, 14-18
 - compiled, 8-11
 - shared SQL areas and, 14-18
 - records, 8-10
 - stored procedures, 1-5, 6-3, 8-2, 8-3
- PMON background process, 15-8
- pragmas, PL/SQL, 10-12
- precompilers, 8-1
 - embedded SQL, 7-9
- predicates, SQL, 3-6
 - SQL
 - predicates, 7-2
- primary key constraints, 5-2
- primary keys, 2-9, 3-2
 - constraints, 5-5
 - hash clusters, 2-26
 - index-organized tables, 2-6
- private SQL areas, 14-16
 - described, 14-16
 - how managed, 14-6
 - parsing and, 7-16
- private synonyms, 4-23
- privileges, 6-1, 7-9
 - administrator, 13-6
 - granting, 7-3
 - PL/SQL procedures and, 8-4
 - revoking, 7-3
- procedures
 - advantages, 8-3
 - execution, 8-5, 8-11
 - external, 8-11

- memory allocation, 8-3
- security, 8-4
- shared SQL areas and, 14-18
- stored procedures, 1-5, 6-3, 8-2, 8-11
- process monitor process (PMON)
 - described, 15-8
- processes, 15-1
 - archiver (ARC*n*), 15-12
 - background, 15-7
 - checkpoints and, 15-9
 - client, 15-3
 - dedicated server, 7-18, 16-14
 - distributed transaction resolution, 15-11
 - job queue, 15-12
 - listener, 16-6
 - shared servers and, 16-13
 - log writer (LGWR), 15-9
 - multiple-process Oracle, 15-1
 - Oracle, 15-6
 - parallel execution coordinator, 15-15
 - parallel execution servers, 15-15
 - process monitor (PMON), 15-8
 - recoverer (RECO), 15-11
 - server, 15-6
 - shared, 16-13, 16-14
 - shared server, 16-11
 - client requests and, 16-12
 - structure, 15-1
 - system monitor (SMON), 15-8
 - user
 - recovery from failure of, 15-8
 - sharing server processes, 16-13
- processing
 - parallel SQL, 15-14
- program global area (PGA), 14-2
 - shared server, 16-14
 - shared servers, 16-14
- program interface, 16-16
 - Oracle side (OPI), 16-16
 - structure of, 16-16
 - user side (UPI), 16-16
- program units, 8-2
 - shared pool and, 14-18
- programming, server-side, 8-2
- pseudocode
 - triggers, 8-21
- pseudocolumns, 2-13, 3-21
- public synonyms, 4-23

Q

- queries
 - blocks, 7-7
 - definition, 7-5
 - implicit, 7-7, 9-7
 - in DML, 7-4
 - parallel processing, 15-14
 - query blocks, 7-12
 - query transformer, 7-11
 - SQL language and, 7-1

- stored, 4-12
 - subqueries, 4-13, 7-4, 7-7
- query blocks, 7-12
- query plans, 7-19
 - partitioning and, 4-1
- query rewrite, 4-19, 4-21
- query transformer, 4-19

R

- range partitions, 4-2
- RDBMS (relational database management system), 1-2
- read committed isolation, 9-6
- read consistency. *See* multiversion read consistency
- read uncommitted, 9-6
- read-only isolation level, 9-11
- Real Application Clusters
 - cache fusion, 9-3
 - system monitor process and, 15-8
 - threads of online redo log, 11-12
- records, PL/SQL, 8-10
- recoverer process (RECO), 15-11
 - in-doubt transactions, 10-13
- recovery
 - complete, 18-14
 - database buffers and, 13-12
 - distributed processing in, 15-11
 - instance recovery
 - SMON process, 15-8
 - media, 18-14
 - media recovery
 - dispatcher processes, 16-14
 - process recovery, 15-8
 - required after terminating instance, 13-10
 - rolling back transactions, 13-14
 - tablespace
 - point-in-time, 18-15
- Recovery Manager (RMAN), 6-5
- recursive SQL, 7-23
- redo log files
 - See also* online redo logs
 - available for use, 11-13
 - circular use of, 11-13
 - contents of, 11-15
 - distributed transaction information in, 11-12
 - group members, 11-14
 - groups, defined, 11-14
 - instance recovery use of, 11-12
 - LGWR and the, 11-13
 - members, 11-14
 - multiplexed, 11-14
 - online, defined, 11-12
 - redo entries, 11-15
 - threads, 11-12
- redo logs
 - archiver process (ARC*n*), 15-12
 - buffer management, 15-9
 - committed data, 13-12
 - committing a transaction, 15-9

- log switch
 - archiver process, 15-12
 - log writer process, 14-14, 15-9
 - rolling forward, 13-12
 - writing buffers, 15-9
- redo logs buffer, 14-14
- redo records, 11-15
- REF constraints, 5-3
- referential integrity
 - examples of, 5-12
 - self-referential constraints, 5-12
- refresh
 - incremental, 4-18
 - job queue processes, 15-12
 - materialized views, 4-18
- relational database management system. *See* RDBMS
- replication, Oracle Streams, 4-17, 17-22
 - bi-directional, 17-23
 - hub-and-spoke, 17-23
 - master database, 4-17
 - n-way, 17-23
 - one-way, 17-23
- reserved words, 7-3
- response queues, 16-12
- result cache, 14-20
- result sets, SQL, 2-7, 2-13, 2-15, 4-15, 7-19, 7-22
- RESULT_CACHE clause, 14-20
- results sets, SQL, 7-5
- reverse key indexes, 3-11
- REVOKE statement, 7-3
- rights, definer's and invoker's, 8-4
- roles, 2-2, 6-3, 7-9
- rollback, 10-8, 10-10
 - described, 10-8, 10-10
 - ending a transaction, 10-8, 10-10
 - implicit in DDL, 7-4
 - statement-level, 10-4
 - to a savepoint, 10-8
- rollback segments
 - locks on, 9-26
 - parallel recovery, 13-14
 - use of in recovery, 13-14
- ROLLBACK statement, 10-6
- rollback, statement-level, 9-16
- rollback, transaction, 7-8
- rolling back, 10-1, 10-8, 10-10
- row cache, 14-19
- row chaining, 12-16
- row data (section of data block), 12-8
- row directories, 12-8
- row locks, 9-18
 - concurrency, 9-19
 - storage, 9-20
- row pieces, 2-19
- row source generation, 7-19
- ROWID data type, 2-13
- rowids, 2-19
 - foreign, 2-13
 - index, 3-4
 - logical, 2-13, 3-24

- physical, 2-13
- row migration, 12-16
- scans, 7-13
- universal, 2-13
- rows
 - addresses, 2-19
 - chaining across blocks, 2-19, 12-16
 - clustered, 2-19
 - definition, 2-7
 - format of in data blocks, 12-8
 - locking, 9-18
 - locks on, 9-18
 - migrating to new block, 12-16
 - row set, 7-19
 - row source, 7-19
 - shown in rowids, 12-10
 - storage, 2-19
 - triggers, 8-17

S

- sample schemas, 2-6
- SAVEPOINT statement, 7-8
- savepoints, 7-8, 10-8
 - definition, 10-8
 - implicit, 10-5
 - rolling back to, 10-8
 - SAVEPOINT statement, 10-6
- scans
 - cluster, 7-13
 - fast full index, 3-7
 - full index, 3-6
 - full table, 3-1, 5-9, 7-12, 14-13
 - index, 3-6, 7-13
 - index skip, 3-8
 - range, 2-26
 - rowid, 7-13
 - unique index, 3-8
- schema objects
 - definitions, 6-1, 7-3
 - dependencies, 2-4, 4-13
 - dimensions, 4-21
 - indexes, 3-2
 - introduction, 2-1
 - materialized views, 4-16
 - relationship to data files, 11-7
 - sequences, 4-20
 - storage, 2-3
- schemas, 2-1
- schemas, sample, 2-6
- SCN
 - See* system change numbers
- secondary indexes, 3-23
 - benefits, 3-24
 - physical guesses, 3-24
- SecureFiles, 19-13
- security
 - administrator privileges, 13-6
 - auditing, 17-5
 - definer's rights, 8-4

- program interface enforcement of, 16-16
- views, 4-12
- segment advisor, 18-22
- segments, 12-21
 - data, 12-21
 - defined, 12-2
 - index, 3-17, 3-20
 - overview of, 12-21
 - table storage, 2-18
 - temporary, 2-16, 12-23
 - allocating, 12-23
- select lists, SQL, 7-5
- SELECT statement, 7-4
- selectivity, 3-7, 3-19
- self-referential integrity constraints, 5-7
- sequences
 - concurrent access, 4-20
 - definition, 4-20
 - surrogate keys, 5-5
- serializability, transactional, 9-1
- serialization isolation level, 9-8
- server parameter file
 - startup, 13-6
- server processes, 15-6
 - listener process, 16-6
- servers
 - client/server architecture, 16-1
 - shared
 - architecture, 15-2, 16-11
 - processes of, 16-11, 16-14
- server-side programming, 8-2
 - overview, 8-1
- service names, 16-6
- service oriented architecture, 1-11, 16-5
- SERVICE_NAMES parameter, 16-9
- session control statements, 7-8
- sessions, 7-8
 - connections contrasted with, 15-4
 - defined, 15-4
 - memory allocation in the large pool, 14-21
 - sequence generation in, 4-20
- SET CONSTRAINT statement, 7-8
- SET TRANSACTION statement, 7-8, 10-3
- SGA (system global area)
 - allocating, 13-6
 - contents of, 14-8
 - data dictionary cache, 6-5, 14-19
 - database buffer cache, 14-9
 - large pool, 14-21
 - redo log buffer, 10-8, 14-14
 - rollback segments and, 10-8
 - shared and writable, 14-8
 - shared pool, 8-3, 14-15
 - variable parameters, 13-15
- share DDL locks, 9-24
- share locks, 9-15
- shared pool, 8-11, 14-15, 14-19
 - allocation of, 14-19
 - check, 7-17
 - dependency management and, 14-19
 - described, 14-15
 - flushing, 14-19
 - latches, 9-25
 - parse locks, 9-24
 - row cache and, 14-19
- shared server
 - described, 15-2
 - dispatcher processes, 16-13
 - Oracle Net Services or SQL*Net V2
 - requirement, 16-13
 - processes, 16-14
 - processes needed for, 16-11
 - restricted operations in, 16-14
 - session memory in the large pool, 14-21
- shared server processes (Snnm), 16-14
 - described, 16-14
- shared SQL areas, 4-14, 7-17, 14-15, 14-16, 14-19
 - dependency management and, 14-19
 - described, 14-16
 - parse locks, 9-24
 - procedures, packages, triggers and, 14-18
- shutdown, 13-8, 13-10
 - abnormal, 13-10
 - prohibited by dispatcher processes, 16-14
 - steps, 13-8
- SHUTDOWN ABORT statement, 13-10
- Simple Object Access Protocol (SOAP). *See* SOAP
- simple triggers, 8-18
- single-level partitioning, 4-2
- SMON background process, 15-8
- SOA, 1-11, 16-5
- SOAP (Simple Object Access Protocol), 1-11
- soft parsing, 7-17
- software code areas, 14-23
- space management
 - extents, 12-18
 - PCTFREE, 12-12
 - row chaining, 12-16
 - segments, 12-21
- SQL, 7-1, 7-3
 - conditions, 7-2, 7-5
 - data definition language (DDL), 7-3
 - data manipulation language (DML), 7-4
 - dictionary cache locks, 9-26
 - dynamic SQL, 8-10
 - embedded, 7-1, 7-9
 - executable, 10-3
 - execution, 7-20, 10-4
 - expressions, 3-3, 7-5
 - functions, 2-13, 7-2
 - IDs, 7-17
 - implicit queries, 7-7
 - interactive, 7-1
 - memory allocation for, 14-17
 - operators, 7-2, 7-5
 - optimization, 7-19
 - Oracle, 7-2
 - overview, 7-1
 - parallel execution, 15-14
 - parsing, 7-16

- PL/SQL and, 8-2
- predicates, 3-6
- recursive, 7-23
- reserved words, 7-3
- result sets, 2-7, 2-13, 2-15, 4-15, 7-5, 7-19, 7-22
- select lists, 7-5
- session control statements, 7-8
- stages of processing, 7-15
- standards, 7-2
- statements, 7-3
- subqueries, 4-13, 7-7
- system control statements, 7-9
- transaction control statements, 7-8
- transactions, 10-1
- transactions and, 10-10
- types of statements, 7-3
- SQL areas
 - private, 14-16
 - shared, 14-16
- SQL Plan Management (SPM), 7-12
- SQL tuning advisor, 18-22
- SQL*Plus
 - alert log, 13-21
 - executing a package, 8-8
 - lock and latch monitors, 9-25
 - SQL statements, 7-2
- SQLJ standard, 8-16
- standards
 - ANSI/ISO, 7-2
 - isolation levels, 9-5
- startup, 13-1
 - prohibited by dispatcher processes, 16-14
- statement-level atomicity, 10-4
- statement-level read consistency, 9-2
- statement-level rollback, 9-16, 10-4
- statements, SQL, 7-3
- statistics, 2-18, 6-5, 7-10, 7-19, 14-19
 - ASH, 18-23
 - AWR, 18-21
 - definition, 7-13
 - gathering for optimizer, 18-22
 - Java-related, 14-22
 - join order, 7-7
 - undo retention, 12-33
- storage
 - logical structures, 12-30
 - nulls, 2-9
 - triggers, 8-21, 8-22
- STORAGE clause
 - using, 12-20
- storage parameters
 - setting, 12-20
- stored procedures. *See* procedures
- Structured Query Language (SQL), 7-1
- structures
 - locking, 9-24
 - logical, 12-1
 - data blocks, 12-1, 12-6
 - extents, 12-1, 12-18
 - segments, 12-1, 12-21
 - tablespaces, 11-1, 12-30
 - physical
 - control files, 11-10
 - data files, 11-1
 - processes, 15-1
- subprograms, PL/SQL. *See* procedures
- subqueries, 4-13, 7-4, 7-7
- summaries, 4-16
- surrogate keys, 5-5
- synonyms
 - constraints indirectly affect, 5-13
 - data dictionary views, 6-5
 - definition, 4-22
 - private, 4-23
 - public, 4-23, 6-4
 - securability, 4-23
- SYS user name, 2-5
 - data dictionary tables, 6-4
 - V\$ views, 6-6
- SYSDBA privilege, 13-6
- SYSOPER privilege, 13-6
- system change numbers
 - definition, 9-4
 - when assigned, 11-15
- system change numbers (SCN), 10-5
 - committed transactions, 10-10
 - defined, 10-10
- system control statements, 7-9
- system global area. *See* SGA
- system locks, 9-25
 - internal, 9-26
 - latches, 9-25
 - mutexes, 9-25
- system monitor process (SMON), 15-8
 - defined, 15-8
 - Real Application Clusters and, 15-8
 - rolling back transactions, 13-14
- SYSTEM tablespace, 6-4
 - data dictionary stored in, 12-32
 - online requirement of, 12-35
- SYSTEM user name, 2-5

T

- table clusters
 - cluster keys, 2-22
 - definition, 2-22
 - hash clusters, 2-25
 - indexed, 2-23
 - scans, 7-13
- tables
 - base, 4-13, 6-4
 - characteristics, 2-8
 - clustered, 2-22
 - compression, 2-19, 4-7
 - definition, 2-2
 - detail, 4-16
 - dimension, 4-21
 - directories, 12-8
 - DUAL, 6-4

- dynamic partitioning, 15-15
- external, 2-16
- fact, 4-21
- full table scans, 3-1
- heap-organized, 2-3, 3-20
- index-organized, 3-20, 3-24, 4-12
- integrity constraints, 5-1
- joins, 3-15
- key-preserved, 4-15
- master, 4-16, 4-17
- normalized or denormalized, 4-22
- object, 2-15
- overview, 2-1
- partitions, 4-7
- permanent, 2-7
- storage, 2-18
- temporary, 2-15, 12-23
- transaction, 10-3
- truncating, 7-3
- views of, 4-12
- virtual, 6-6
- tables, base, 4-12
- tables, external, 2-7
- tables, temporary, 2-7
- tablespace point-in-time recovery, 18-15
- tablespaces, 12-30
 - described, 12-30
 - dictionary managed, 12-6
 - locally managed, 12-3
 - locks on, 9-26
 - offline, 12-35
 - online, 12-35
 - overview of, 12-30
 - recovery, 18-15
 - schema objects, 2-3
 - space allocation, 12-2
 - SYSTEM, 6-4
 - used for temporary segments, 12-23
- tasks, 15-1
- temp files, 11-8
- temporary segments, 2-16, 12-23
 - allocating, 12-23
 - allocation for queries, 12-23
- temporary tables, 2-7, 2-15
- threads
 - online redo log, 11-12
- time zones, 2-13
 - in date/time columns, 2-12
- TIMESTAMP data type, 2-12
- TO_CHAR function, 2-14
- TO_DATE function, 2-12, 2-14
- trace files
 - LGWR trace file, 15-10
- transaction control statements, 7-8
- transaction tables, 9-5, 10-3
 - reset at recovery, 15-8
- transaction-level read consistency, 9-2
- transactions, 10-1
 - ACID properties, 10-1
 - active, 10-7
 - assigning system change numbers, 10-10
 - autonomous, 10-11
 - within a PL/SQL block, 10-12
 - beginning, 10-3
 - blocking, 9-7
 - committing, 10-10, 15-9
 - group commits, 15-10
 - conflicting writes, 9-7
 - deadlocks, 9-16
 - deadlocks and, 10-5
 - definition, 10-1
 - distributed, 10-7, 10-12
 - resolving automatically, 15-11
 - DML statements, 7-5
 - ending, 10-3
 - enqueued, 10-9
 - in-doubt
 - resolving automatically, 10-13
 - interested transaction lists (ITLs), 9-5
 - isolation levels, 9-5, 9-8, 9-11
 - isolation of, 9-5
 - naming, 10-7
 - read consistency, 9-2
 - rolling back, 10-8, 10-10
 - partially, 10-8
 - savepoints in, 10-8
 - serializability, 9-1
 - setting properties, 7-8
 - terminating the application and, 10-4
 - transaction control statements, 7-8
 - transaction history, 9-5
 - transaction ID, 10-1, 10-3
 - transaction tables, 9-5
- triggers, 8-2
 - cascading, 8-17
 - components of, 8-18
 - compound, 8-18
 - effect of rollbacks, 10-5
 - firing (executing), 8-21
 - privileges required, 8-21
 - INSTEAD OF, 8-18
 - overview, 8-16
 - restrictions, 8-19
 - row, 8-17
 - shared SQL areas and, 14-18
 - simple, 8-18
 - statement, 8-17
 - storage of, 8-21
 - timing, 8-18
 - UNKNOWN does not fire, 8-19
 - uses of, 8-17
- TRUNCATE statement, 7-3
- two-phase commit
 - transaction management, 10-13

U

- undo management, automatic, 12-24
- undo segments, 10-3
 - read consistency, 9-3

- undo tablespaces, 12-33
 - undo retention period, 9-11
- Unicode, 2-10
- unique indexes, 3-4
- unique key constraints, 5-2, 5-3
 - composite keys, 5-4
 - NOT NULL constraints and, 5-4
- unplanned downtime
 - avoiding downtime during, 17-7
- update no action constraint, 5-8
- UPDATE statement, 7-4
- updates
 - lost, 9-7
 - updatability of views, 4-15, 8-18
 - updatable join views, 4-15
- updating tables
 - with parent keys, 9-21
- UROWID data type, 2-13
- user program interface (UPI), 16-16
- USER_ views, 6-4
- users, database, 2-2
 - authentication, 17-3
 - names, 6-1
 - sessions and connections, 15-4
 - privileges, 2-1
 - temporary tablespaces, 12-23
- UTL_HTTP package, 8-6

V

- V\$ views, 6-6
- VARCHAR2 data type, 2-10
- variables
 - embedded SQL, 7-9
- views, 4-12
 - base tables, 4-12
 - constraints indirectly affect, 5-13
 - data access, 4-14
 - data dictionary
 - updatable columns, 4-15
 - fixed views, 6-6
 - indexes, 4-14
 - INSTEAD OF triggers, 8-18
 - integrity constraints, 4-14
 - materialized, 4-16
 - object, 4-16
 - schema object dependencies, 4-13
 - storage, 4-13
 - updatability, 4-15
 - uses, 4-12
 - V\$, 6-5
- virtual columns, 2-7, 2-18, 3-19

W

- warehouse
 - materialized views, 4-16
- Web services, 1-11, 16-5
- write-ahead, 15-9

X

- XA
 - session memory in the large pool, 14-21
- XMLType data type, 19-11

