Oracle9*i*

Database Concepts

Release 1 (9.0.1)

July 2001

Part No. A88856-02

ORACLE

Oracle9*i* Database Concepts, Release 1 (9.0.1)

Part No. A88856-02

Primary Author: Lenore McGee Luscher

Contributors: Patrick Amor, Lance Ashdown, Cathy Baird, Lee Barton, Mehul Bastawala, Mark Bauer, Ruth Baylis, Neerja Bhatt, Allen Brumm, Ted Burroughs, Chandra Chandrasekar, Gary Chen, Eugene Chong, Michele Cyran, Amit Ganesh, John Haydu, Brian Hirano, Thuvan Hoang, Bob Jenkins, Vishy Karra, Ravikanth Kasamsetty, Susan Kotsovolos, Julie Laffrenzen, Tirthankar Lahiri, Paul Lane, Simon Law, Yunrui Li, Diana Lorentz, Karen McKeen, Ben Meng, Magdi Morsi, Ari Mozes, Sreedhar Mukkamalla, Subramanian Muralidhar, Ravi Murthy, Sujatha Muthulingam, Bhagat Nainani, Gary Ngai, Jeffrey Olkin, Kant Patel, Ananth Raghavan, Jack Raitto, Beck Reitmeyer, Ann Rhee, John Russell, Vivian Schupmann, Ravi Shankar, Lei Sheng, Mark Smith, Richard Smith, Ekrem Soylemez, Marie St. Gelais, James Stenoish, Debbie Steiner, Sankar Subramanian, Seema Sundara, Bob Thome, Ashish Thusoo, Anh-Tuan Tran, Stephen J. Vivian, Daniel Wong, Adiel Yoaz, Qin Yu

Graphic Designer: Valarie Moore

Contents

2 Oracle Server Features

7 Distributed Processing

8 Memory Architecture

Part IV The Object-Relational DBMS

11 Schema Objects

13 System-Provided Datatypes

14 User-Defined Datatypes

15 Object Views

Part V Data Access

16 SQL, PL/SQL, and Java

17 Transaction Management

18 Triggers

19 Dependencies Among Schema Objects

Part VI Parallel SQL and Direct-Load INSERT

20 Parallel Execution of SQL Statements

23 Data Integrity

24 Controlling Database Access

25 Privileges, Roles, and Security Policies

26 Auditing

Send Us Your Comments

Oracle9*i* Database Concepts, Release 1 (9.0.1)

Part No. A88856-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This manual describes all features of the Oracle server, an object-relational database management system. It describes how the Oracle server functions and lays a conceptual foundation for much of the practical information contained in other Oracle server manuals. Information in this manual applies to the Oracle server running on all operating systems.

Oracle9*i* and Oracle9*i* Enterprise Edition

Oracle9i Database Concepts contains information that describes the functionality of the Oracle9*i* (also known as the standard edition) and the Oracle9*i* Enterprise Edition products. Oracle9*i* and Oracle9*i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use application failover, you must have the Enterprise Edition with the Real Application Clusters option.

For information about the differences between Oracle9*i* and the Oracle9*i* Enterprise Edition and the features and options that are available to you, see *Oracle9i Database New Features.*

This preface contains these topics:

- Audience
- Organization
- Related Documentation
- Conventions
- Documentation Accessibility

Audience

This manual is intended for database administrators, system administrators, and database application developers.

To use this document, you need to know the following:

- Relational database concepts in general
- Concepts and terminology in **Chapter 1, "Introduction to the Oracle Server"**
- The operating system environment under which you are running Oracle

Organization

This document contains:

Part I: What Is Oracle?

Chapter 1, "Introduction to the Oracle Server"

This chapter provides an overview of the concepts and terminology you need for understanding the Oracle data server. You should read this overview before using the detailed information in the remainder of this manual.

Chapter 2, "Oracle Server Features"

This chapter provides an overview of some of the features of the Oracle server.

Part II: Database Structures

Chapter 3, "Data Blocks, Extents, and Segments"

This chapter discusses how data is stored and how storage space is allocated for and consumed by various objects within an Oracle database.

Chapter 4, "Tablespaces, Datafiles, and Control Files"

This chapter discusses how physical storage space in an Oracle database is divided into logical divisions called tablespaces. It also discusses the physical operating system files associated with tablespaces (datafiles) and files used in recovery (control files).

Chapter 5, "The Data Dictionary"

This chapter describes the data dictionary, which is a set of reference tables and views that contain read-only information about an Oracle database.

Part III: The Oracle Instance

Chapter 6, "Database and Instance Startup and Shutdown"

This chapter describes an Oracle instance and explains how the database administrator can control the accessibility of an Oracle database system. This chapter also describes the parameters that control how the database operates.

Chapter 7, "Distributed Processing"

This chapter discusses distributed processing environments in which the Oracle data server can operate.

Chapter 8, "Memory Architecture"

This chapter describes the memory structures used by an Oracle database system.

Chapter 9, "Process Architecture"

This chapter describes the process architecture of an Oracle instance and the different process configurations available for Oracle.

Chapter 10, "Database Resource Management"

This chapter describes how the Database Resource Manager can be used to control resource use.

Part IV: The Object-Relational DBMS

Chapter 11, "Schema Objects"

This chapter describes the database objects that can be created in the domain of a specific user (a schema), including tables, views, numeric sequences, and synonyms. Optional structures that make data retrieval more efficient, including indexes, materialized views, dimensions, and clusters, are also described.

Chapter 12, "Partitioned Tables and Indexes"

This chapter describes how partitioning can be used to split large tables and indexes into more manageable pieces.

Chapter 13, "System-Provided Datatypes"

This chapter describes the types of relational data that can be stored in an Oracle database table, such as fixed- and variable-length character strings, numbers, dates, and binary large objects (BLOBs).

Chapter 14, "User-Defined Datatypes"

This chapter gives an overview of the object extensions that Oracle provides.

Chapter 15, "Object Views"

This chapter describes the extensions to views provided by the Oracle data server.

Part V: Data Access

Chapter 16, "SQL, PL/SQL, and Java"

This chapter briefly describes SQL (Structured Query Language), the language used to communicate with Oracle, as well as PL/SQL, the Oracle procedural language extension to SQL. It also discusses the procedural language constructs called procedures, functions, and packages, which are PL/SQL program units that are stored in the database.

Chapter 17, "Transaction Management"

This chapter defines the concept of transactions and explains the SQL statements used to control them. Transactions are logical units of work that are executed together as a unit.

Chapter 18, "Triggers"

This chapter discusses triggers, which are procedures written in PL/SQL, Java, or C that execute (fire) implicitly whenever a table or view is modified or when some user actions or database system actions occur.

Chapter 19, "Dependencies Among Schema Objects"

This chapter explains how Oracle manages the dependencies for objects such as procedures, packages, triggers, and views.

Part VI: Parallel SQL and Direct-Path INSERT

Chapter 20, "Parallel Execution of SQL Statements"

This chapter describes parallel execution of SQL statements (queries, DML, and DDL statements) and explains the rules for parallelizing SQL statements.

Chapter 21, "Direct-Path INSERT"

This chapter describes the Oracle direct-path `INSERT` feature for serial or parallel inserts, and the `NOLOGGING` clause.

Part VII: Data Protection

Chapter 22, "Data Concurrency and Consistency"

This chapter explains how Oracle provides concurrent access to and maintains the accuracy of shared information in a multiuser environment. It describes the automatic mechanisms that Oracle uses to guarantee that the concurrent operations of multiple users do not interfere with each other.

Chapter 23, "Data Integrity"

This chapter discusses data integrity and the declarative integrity constraints that you can use to enforce data integrity.

Chapter 24, "Controlling Database Access"

This chapter describes how to control user access to data and database resources.

Chapter 25, "Privileges, Roles, and Security Policies"

This chapter discusses security at the system and schema object levels.

Chapter 26, "Auditing"

This chapter discusses how the Oracle auditing feature tracks database activity.

Appendix A

This appendix lists all the operating system specific references within this manual.

Glossary

The glossary defines terms used in this manual.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Database Migration* for information about upgrading and migration
- *Oracle9i Database Administrator's Guide* for information about how to administer the Oracle server
- *Oracle9i Application Developer's Guide - Fundamentals* for information about developing Oracle database applications
- *Oracle9i Database Performance Methods* for information about optimizing performance of an Oracle database
- *Oracle9i Data Warehousing Guide* for information about data warehousing and business intelligence

Many books in the documentation set use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://technet.oracle.com/membership/index.htm`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://technet.oracle.com/docs/index.htm`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading

technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

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

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Part I

What Is Oracle?

Part I provides an overview of Oracle server concepts and terminology. It contains the following chapters:

- Chapter 1, "Introduction to the Oracle Server"
- Chapter 2, "Oracle Server Features"

1

Introduction to the Oracle Server

This chapter provides an overview of the Oracle server. The topics include:

- Introduction to Databases and Information Management
- Database Structure and Space Management
- Memory Structure and Processes
- The Object-Relational Model for Database Management
- Data Concurrency and Consistency Overview
- Startup and Shutdown Operations

> **Note:** This chapter contains information relating to both Oracle9*i* and the Oracle9*i* Enterprise Edition. Some of the features and options documented in this chapter are available only if you have purchased the Oracle9*i* Enterprise Edition. See *Oracle9i Database New Features* for information about the differences between Oracle9*i* and the Oracle9*i* Enterprise Edition.

Introduction to Databases and Information Management

A database server is the key to solving the problems of information management. In general, a server must reliably manage a large amount of data in a multiuser environment so that many users can concurrently access the same data. All this must be accomplished while delivering high performance. A database server must also prevent unauthorized access and provide efficient solutions for failure recovery.

The Oracle server provides solutions with the following features:

Client/server environments (distributed processing)	To take full advantage of a given computer system or network, Oracle enables processing to be split between the database server and the client application programs. The computer running the database management system handles all of the database server responsibilities while the workstations running the database applications concentrate on the interpretation and display of data.
Large databases and space management	Oracle supports the largest databases, which can contain terabytes of data. To make efficient use of expensive hardware devices, Oracle enables full control of space usage.
Many concurrent database users	Oracle supports large numbers of concurrent users executing a variety of database applications operating on the same data. It minimizes data contention and guarantees data concurrency.
Connectibility	Oracle software enables different types of computers and operating systems to share information across networks.
High transaction processing performance	Oracle maintains the preceding features with a high degree of overall system performance. Database users do not experience slow processing performance.
High availability	At some sites, Oracle works 24 hours a day with no down time to limit database throughput. Normal system operations such as database backup and partial computer system failures do not interrupt database use.
Controlled availability	Oracle can selectively control the availability of data, at the database level and below. For example, an administrator can disallow use of a specific application so the application's data can be reloaded without affecting other applications.

Openness, industry standards	Oracle adheres to industry accepted standards for the data access language, operating systems, user interfaces, and network communication protocols. It is an open system that protects a customer's investment. Oracle also supports the Simple Network Management Protocol (SNMP) standard for system management. This protocol enables administrators to manage heterogeneous systems with a single administration interface.
Manageable security	To protect against unauthorized database access and use, Oracle provides security features to limit and monitor data access. These features make it easy to manage even the most complex design for data access.
Database enforced integrity	Oracle enforces data integrity, business rules that dictate the standards for acceptable data. This reduces the costs of coding and managing checks in many database applications.
Portability	Oracle software works under different operating systems. Applications developed for Oracle can be ported to any operating system with little or no modification.
Compatibility	Oracle software is compatible with industry standards, including most industry standard operating systems. Applications developed for Oracle can be used on virtually any system with little or no modification.
Distributed systems	For networked, distributed environments, Oracle combines the data physically located on different computers into one logical database that can be accessed by all network users. Distributed systems have the same degree of user transparency and data consistency as nondistributed systems, yet they receive the advantages of local database management. Oracle also offers the heterogeneous option that enables users to access data on some non-Oracle databases transparently.

Replicated environments	Oracle software lets you replicate groups of tables and their supporting objects to multiple sites. Oracle supports replication of both data- and schema-level changes to these sites. Oracle's flexible replication technology supports basic primary site replication as well as advanced dynamic and shared-ownership models.

The following sections provide a comprehensive overview of the Oracle architecture. Each section describes a different part of the overall architecture.

The Oracle Server

The Oracle server is an object-relational database management system that provides an integrated approach to information management. An Oracle server consists of an Oracle database and an Oracle server instance.

An Oracle Instance

Every time a database is started, a system global area (SGA) is allocated and Oracle background processes are started. The system global area is an area of memory used for database information shared by the database users. The combination of the background processes and memory buffers is called an Oracle **instance**.

An Oracle instance has two types of processes: user processes and Oracle processes.

- A **user process** executes the code of an application program (such as an Oracle Forms application) or an Oracle Tool (such as Oracle Enterprise Manager).
- **Oracle processes** are server processes that perform work for the user processes and background processes that perform maintenance work for the Oracle server.

Oracle9*i* Real Application Clusters: Multiple Instance Systems

> **Note:** Real Application Clusters are available only if you have purchased the Oracle9*i* Enterprise Edition. See *Oracle9i Database New Features* for details about the features and options available with Oracle9*i* Enterprise Edition.

Some hardware architectures (for example, shared disk systems) enable multiple computers to share access to data, software, or peripheral devices. Oracle9*i* with

Real Application Clusters can take advantage of such architecture by running multiple instances that share a single physical database. In most applications, Real Application Clusters enable access to a single database by the users on multiple machines with increased performance.

Real Application Clusters are inherently high availability systems. The clusters that are typical of Real Application Clusters environments can provide continuous service for both planned and unplanned outages.

> **See Also:** *Oracle9i Real Application Clusters Concepts* for more information about Real Application Clusters and high availability

Database Structure and Space Management

An Oracle **database** is a collection of data that is treated as a unit. The purpose of a database is to store and retrieve related information. The database has **logical structures** and **physical structures**. Because the physical and logical structures are separate, the physical storage of data can be managed without affecting the access to logical storage structures.

Logical Database Structures

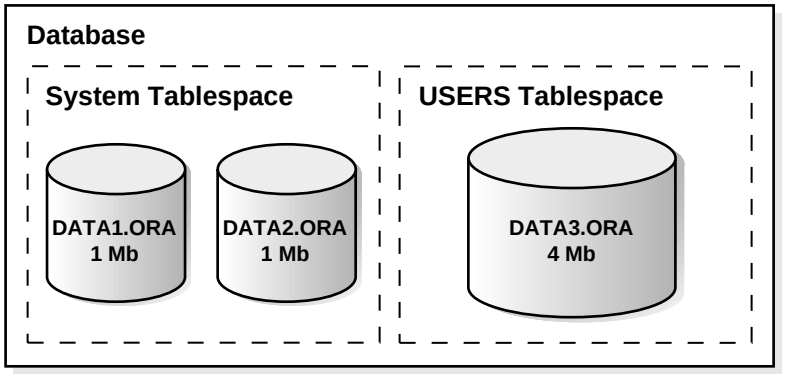
The logical structures of an Oracle database include tablespaces, schema objects, data blocks, extents, and segments.

Tablespaces

A database is divided into logical storage units called **tablespaces**, which group related logical structures together. For example, tablespaces commonly group all of an application's objects to simplify some administrative operations.

Databases, Tablespaces, and Datafiles The relationship among databases, tablespaces, and datafiles (datafiles are described in the next section) is illustrated in Figure 1–1.

Figure 1–1 Databases, Tablespaces, and Datafiles



This figure illustrates the following:

- Each database is logically divided into one or more tablespaces.
- One or more datafiles are explicitly created for each tablespace to physically store the data of all logical structures in a tablespace.
- The combined size of a tablespace's datafiles is the total storage capacity of the tablespace (SYSTEM tablespace has 2 MB storage capacity while USERS tablespace has 4 MB).
- The combined storage capacity of a database's tablespaces is the total storage capacity of the database (6 MB).

Online and Offline Tablespaces Description A tablespace can be **online** (accessible) or **offline** (not accessible). A tablespace is normally online so that users can access the information within the tablespace. However, sometimes a tablespace is taken offline to make a portion of the database unavailable while allowing normal access to the remainder of the database. This makes many administrative tasks easier to perform.

Schemas and Schema Objects

A **schema** is a collection of database objects. **Schema objects** are the logical structures that directly refer to the database's data. Schema objects include such structures as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links.

Note: There is no relationship between a tablespace and a schema. Objects in the same schema can be in different tablespaces, and a tablespace can hold objects from different schemas.

See Also: "Description of Schemas and Schema Objects" on page 1-24 for more information about schema objects

Description of Data Blocks, Extents, and Segments

Oracle enables fine-grained control of disk space use through the logical storage structures, including data blocks, extents, and segments.

See Also: Chapter 3, "Data Blocks, Extents, and Segments"

Oracle 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 of physical database space on disk. The standard block size is specified by the initialization parameter `DB_BLOCK_SIZE`. In addition, Oracle9*i*, Release 1 (9.0.1), permits specification of up to five nonstandard block sizes. A database uses and allocates free database space in Oracle data blocks.

See Also: "Nonstandard Block Sizes" on page 4-13 for information about using nonstandard block sizes

Extents The next level of logical database space is called an **extent**. An extent is a specific number of contiguous data blocks, obtained in a single allocation, used to store a specific type of information.

Segments The level of logical database storage above an extent is called a **segment**. A segment is a set of extents allocated for a certain logical structure. For example, the different types of segments include:

Data segment	Each nonclustered table has a data segment. All of the table's data is stored in the extents of its data segment. For a partitioned table, each partition has a data segment. Each cluster has a data segment. The data of every table in the cluster is stored in the cluster's data segment.
Index segment	Each index has an index segment that stores all of its data. For a partitioned index, each partition has an index segment.

Rollback segment If you are operating in manual undo management mode, one or more rollback segments for a database are created by the database administrator to temporarily store undo information.

The information in a rollback segment is used:

- To generate read-consistent database information
- During database recovery
- To roll back uncommitted transactions for users

If you are operating in automatic undo management mode, the database server manages undo space using tablespaces. See "Automatic Undo Management" on page 2-29 for more information about using automatic undo management mode.

Temporary segment Temporary segments are created by Oracle when a SQL statement needs a temporary work area to complete execution. When the statement finishes execution, the temporary segment's extents are returned to the system for future use.

Oracle dynamically allocates space when the existing extents of a segment become full. Therefore, when the existing extents of a segment are full, Oracle allocates another extent for that segment as needed. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on disk.

See Also:

- "Read Consistency" on page 1-32
- "Database Backup and Recovery Overview" on page 2-25

Physical Database Structures

The following sections explain the physical database structures of an Oracle database, including datafiles, redo log files, and control files.

Datafiles

Every Oracle database has one or more physical **datafiles**. The datafiles contain all the database data. The data of logical database structures, such as tables and indexes, is physically stored in the datafiles allocated for a database.

The characteristics of datafiles are:

- A datafile can be associated with only one database.
- Datafiles can have certain characteristics set to let them automatically extend when the database runs out of space.
- One or more datafiles form a logical unit of database storage called a tablespace, as discussed earlier in this chapter.

Data in a datafile is read, as needed, during normal database operation and stored in the memory cache of Oracle. For example, assume that a user wants to access some data in a table of a database. If the requested information is not already in the memory cache for the database, it is read from the appropriate datafiles and stored in memory.

Modified or new data is not necessarily written to a datafile immediately. To reduce the amount of disk access and increase performance, data is pooled in memory and written to the appropriate datafiles all at once, as determined by the database writer (DBW*n*) background process of Oracle.

> **See Also:** "Memory Structure and Processes" on page 1-12 for more information about Oracle's memory and process structures and the algorithm for writing database data to the datafiles

Redo Log Files

Every Oracle database has a set of two or more **redo log files**. The set of redo log files for a database is collectively known as the database's redo log. A redo log is made up of redo entries (also called **redo records**).

The primary function of the redo log is to record all changes made to data. If a failure prevents modified data from being permanently written to the datafiles, the changes can be obtained from the redo log so work is never lost.

Redo log files are critical in protecting a database against failures. To protect against a failure involving the redo log itself, Oracle allows a **multiplexed redo log** so that two or more copies of the redo log can be maintained on different disks.

The information in a redo log file is used only to recover the database from a system or media failure that prevents database data from being written to a database's datafiles.

For example, if an unexpected power outage terminates database operation, data in memory cannot be written to the datafiles and the data is lost. However, any lost data can be recovered when the database is opened, after power is restored. By

applying the information in the most recent redo log files to the database's datafiles, Oracle restores the database to the time at which the power failure occurred.

The process of applying the redo log during a recovery operation is called **rolling forward**.

> **See Also:** "Database Backup and Recovery Overview" on page 2-25 for more information about redo log files

Control Files

Every Oracle database has a **control file**. A control file contains entries that specify the physical structure of the database. For example, it contains the following types of information:

- Database name
- Names and locations of datafiles and redo log files
- Time stamp of database creation

Like the redo log, Oracle lets the control file be multiplexed for protection of the control file.

The Use of Control Files Every time an instance of an Oracle database is started, its control file is used to identify the database and redo log files that must be opened for database operation to proceed. If the physical makeup of the database is altered (for example, a new datafile or redo log file is created), the control file is automatically modified by Oracle to reflect the change.

A control file is also used if database recovery is necessary.

> **See Also:** "Database Backup and Recovery Overview" on page 2-25 for more information about the use of control files in database recovery

Structured Query Language (SQL)

SQL (pronounced SEQUEL) is the programming language that defines and manipulates the database. SQL databases are relational databases, which means data is stored in a set of simple relations. A database can have one or more tables. Each table has columns and rows. Oracle stores each row of a database table containing data for less than 256 columns as one or more row pieces. A table that has an employee database, for example, can have a column called employee number and each row in that column is an employee's employee number.

You can define and manipulate data in a table with SQL statements. You use SQL's data definition language (DDL) statements to set up the data. DDL statements include statements for creating and altering databases and tables.

You can update, delete, or retrieve data in a table with SQL's data manipulation language (DML). DML statements include statements to alter and fetch data. The most common SQL statement is the `SELECT` statement, which retrieves data from the database.

In addition to SQL statements, the Oracle server has a procedural language called PL/SQL. PL/SQL enables programmers to program SQL statements. It lets you control the flow of a SQL program, use variables, and write error-handling procedures.

Data Utilities

There are three utilities for moving a subset of an Oracle database from one database to another: Export, Import, and SQL*Loader.

Export Utility The Export utility transfers data objects between Oracle databases, even if they reside on platforms with different hardware and software configurations. Export extracts the object definitions and table data from an Oracle database and stores them in an Oracle binary-format Export dump file located typically on disk or tape.

Such files can then be copied using FTP or physically transported (in the case of tape) to a different site and used, with the Import utility, to transfer data between databases that are on machines not connected through a network or as backups in addition to normal backup procedures.

When you run Export against an Oracle database, it extracts objects such as tables, followed by their related objects, and then writes them to the Export dump file.

Import Utility The Import utility inserts the data objects extracted from one Oracle database by the Export utility into another Oracle database. Export dump files can be read only by Import.

Import reads the object definitions and table data that the Export utility extracted from an Oracle database and stored in an Oracle binary-format Export dump file located typically on disk or tape.

The Export and Import utilities can also facilitate certain aspects of Oracle Advanced Replication functionality, such as offline instantiation.

> **See Also:** *Oracle9i Replication* for more information about Oracle Advanced Replication

SQL*Loader Utility Export dump files can be read only by the Oracle Import utility. If you need to read load data from ASCII fixed-format or delimited files, you can use the SQL*Loader utility. SQL*Loader loads data from external files into tables in an Oracle database. SQL*Loader accepts input data in a variety of formats, can perform filtering (selectively loading records based upon their data values), and can load data into multiple Oracle database tables during the same load session.

> **See Also:** *Oracle9i Database Utilities* for detailed information about Export, Import, and SQL*Loader

Memory Structure and Processes

This section discusses the memory and process structures used by an Oracle server to manage a database. The architectural features discussed in this section enable the Oracle server to support:

- Many users concurrently accessing a single database
- The high performance required by concurrent multiuser, multi-application database systems

An Oracle server 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 database system.

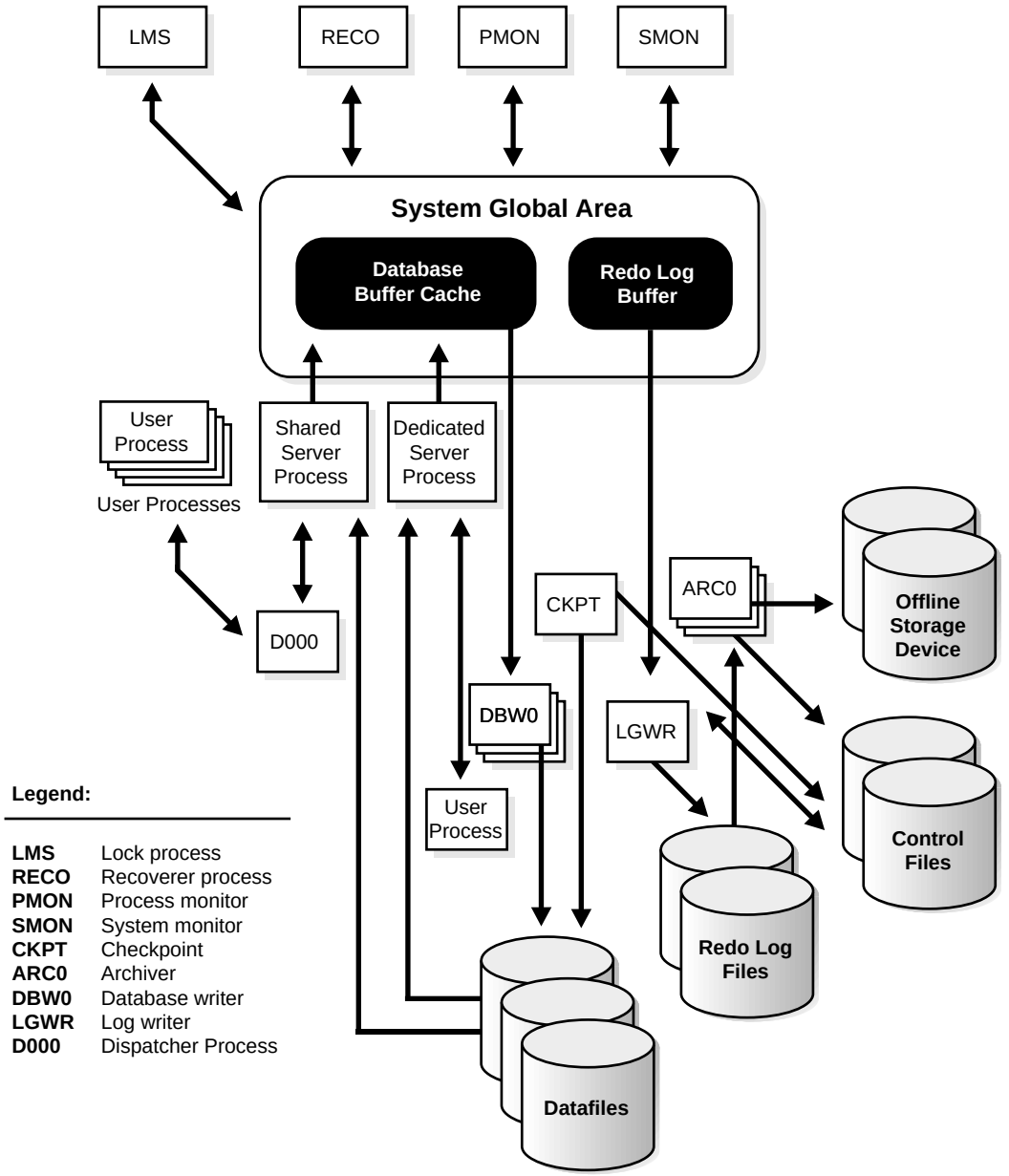
Processes are jobs or tasks that work in the memory of these computers. (Processes are known as "threads" in the NT environment. See "Processes and Threads on Windows NT" on page 1-21.)

Figure 1–2 on page 1-13 shows a typical variation of the Oracle server memory and process structures.

Memory Structures

Oracle creates and uses memory structures to complete several jobs. For example, memory stores program code being executed and data that is shared among users. Two basic memory structures are associated with Oracle: the system global area (which includes the database buffers, redo log buffers, and the shared pool) and the program global areas. The following subsections explain each in detail.

Figure 1–2 Memory Structures and Processes of Oracle



System Global Area

The **System Global Area (SGA)** is a shared memory region that contains data and control information for one Oracle instance. An SGA and the Oracle background processes constitute an Oracle instance.

Oracle allocates the system global area when an instance starts and deallocates it when the instance shuts down. Each instance has its own system global area.

Users currently connected to an Oracle server share the data in the system global area. For optimal performance, the entire system global area should be as large as possible (while still fitting in real memory) to store as much data in memory as possible and minimize disk I/O.

The information stored within the system global area is divided into several types of memory structures, including the database buffers, redo log buffer, and the shared pool. These areas have fixed sizes and are created during instance startup.

See Also:

- "An Oracle Instance" on page 1-4
- "Background Processes" on page 1-16 for more information about the SGA and the Oracle background processes

Database Buffer Cache of the SGA **Database buffers** of the system global area store the most recently used blocks of data. The set of database buffers in an instance is the **database buffer cache**. The buffer cache contains modified as well as unmodified blocks. Because the most recently (and often the most frequently) used data is kept in memory, less disk I/O is necessary and performance is improved.

Redo Log Buffer of the SGA The **redo log buffer** of the system global area stores **redo entries**—a log of changes made to the database. The redo entries stored in the redo log buffers are written to an online redo log file, which is used if database recovery is necessary. The size of the redo log is static.

Shared Pool of the SGA The shared pool is a portion of the system global area that contains shared memory constructs such as shared SQL areas. A shared SQL area is required to process every unique SQL statement submitted to a database. A shared SQL area contains information such as the parse tree and execution plan for the corresponding statement. A single shared SQL area is used by multiple applications that issue the same statement, leaving more shared memory for other uses.

> **See Also:** "SQL Statements" on page 2-11 for more information about shared SQL areas

Large Pool in the SGA The large pool is an optional area in the SGA that provides large memory allocations for Oracle backup and restore operations, I/O server processes, and session memory for the shared server and Oracle XA (used where transactions interact with more than one database).

Statement Handles or Cursors A **cursor** is a handle (a name or pointer) for the memory associated with a specific statement. (The Oracle Call Interface, OCI, refers to these as **statement handles**.) Although most Oracle users rely on automatic cursor handling of Oracle utilities, the programmatic interfaces offer application designers more control over cursors.

For example, in precompiler application development, a cursor is a named resource available to a program and can be used specifically to parse SQL statements embedded within the application. The application developer can code an application so it controls the phases of SQL statement execution and thus improves application performance.

Program Global Area (PGA)

The **Program Global Area (PGA)** is a memory buffer that contains data and control information for a server process. A PGA is created by Oracle when a server process is started. The information in a PGA depends on the Oracle configuration.

Process Architecture

A **process** is a "thread of control" or a mechanism in an operating system that can execute a series of steps. Some operating systems use the terms **job** or **task**. A process normally has its own private memory area in which it runs.

An Oracle server has two general types of processes: user processes and Oracle processes.

User (Client) Processes

A **user process** is created and maintained to execute the software code of an application program (such as a Pro*C/C++ program) or an Oracle tool (such as Oracle Enterprise Manager). The user process also manages the communication with the server processes.

User processes communicate with the server processes through the program interface, which is described in a later section.

Oracle Process Architecture

Oracle processes are called (invoked) by other processes to perform functions on behalf of the invoking process. The different types of Oracle processes and their specific functions are discussed in the following sections.

Server Processes

Oracle creates **server processes** to handle requests from connected user processes. A server process is in charge of communicating with the user process and interacting with Oracle to carry out requests of the associated user process. For example, if a user queries some data that is not already in the database buffers of the system global area, the associated server process reads the proper data blocks from the datafiles into the system global area.

Oracle can be configured to vary the number of user processes for each server process. In a **dedicated server configuration**, a server process handles requests for a single user process. A **shared server configuration** lets many user processes share a small number of server processes, minimizing the number of server processes and maximizing the use of available system resources.

On some systems, the user and server processes are separate, while on others they are combined into a single process. If a system uses the shared server or if the user and server processes run on different machines, the user and server processes must be separate. Client/server systems separate the user and server processes and execute them on different machines.

Background Processes

Oracle creates a set of **background processes** for each instance. They consolidate functions that would otherwise be handled by multiple Oracle programs running for each user process. The background processes asynchronously perform I/O and monitor other Oracle processes to provide increased parallelism for better performance and reliability.

An SGA and the set of Oracle background processes constitute an Oracle instance. Each Oracle instance can use several background processes. The names of these processes are DBW*n*, LGWR, CKPT, SMON, PMON, ARC*n*, RECO, D*nnn*, LMS, Jnnn, and QMN*n*.

See Also:

- "An Oracle Instance" on page 1-4
- "System Global Area" on page 1-14 for more information about the SGA

Database Writer (DBW*n*) The **database writer** writes modified blocks from the database buffer cache to the datafiles. Although one database writer process (DBW0) is sufficient for most systems, you can configure additional processes (DBW1 through DBW9) to improve write performance for a system that modifies data heavily. The initialization parameter `DB_WRITER_PROCESSES` specifies the number of DBW*n* processes.

Because Oracle uses write-ahead logging, DBW*n* does not need to write blocks when a transaction commits. Instead, DBW*n* is designed to perform batched writes with high efficiency. In the most common case, DBW*n* writes only when more data needs to be read into the system global area and too few database buffers are free. The least recently used data is written to the datafiles first. DBW*n* also performs writes for other functions such as checkpointing.

See Also: "Transactions" on page 2-13 for more information about commits

Log Writer (LGWR) The **log writer** writes redo log entries to disk. Redo log entries are generated in the redo log buffer of the SGA, and LGWR writes the redo log entries sequentially into an online redo log file. If the database has a multiplexed redo log, LGWR writes the redo log entries to a group of online redo log files.

Checkpoint (CKPT) At specific times, all modified database buffers in the system global area are written to the datafiles by DBW*n*. This event is called a **checkpoint**. The checkpoint process is responsible for signaling DBW*n* at checkpoints and updating all the datafiles and control files of the database to indicate the most recent checkpoint.

System Monitor (SMON) The **system monitor** performs crash recovery when a failed instance starts up again. In a multiple instance system (one that uses Oracle9*i* Real Application Clusters), the SMON process of one instance can perform instance recovery for other instances that have failed. SMON also cleans up temporary segments that are no longer in use and recovers dead transactions skipped during crash and instance recovery because of file-read or offline errors. These transactions are eventually recovered by SMON when the tablespace or file is brought back

online. SMON also coalesces free extents within the database's dictionary-managed tablespaces to make free space contiguous and easier to allocate.

Process Monitor (PMON) The **process monitor** performs process recovery when a user process fails. PMON is responsible for cleaning up the cache and freeing resources that the process was using. PMON also checks on dispatcher and server processes and restarts them if they have failed.

Archiver (ARC*n*) The **archiver** copies the online redo log files to archival storage after a log switch has occurred. Although a single ARC*n* process (ARC0) is sufficient for most systems, you can specify up to 10 ARC*n* processes by using the dynamic initialization parameter `LOG_ARCHIVE_MAX_PROCESSES`. If the workload becomes too great for the current number of ARC*n* processes, LGWR automatically starts another ARC*n* process up to the maximum of 10 processes. ARC*n* is active only when a database is in ARCHIVELOG mode and automatic archiving is enabled.

> **See Also:** "The Redo Log" on page 2-28 for more information about the archiver

Recoverer (RECO) The **recoverer** is used to resolve distributed transactions that are pending due to a network or system failure in a distributed database. At timed intervals, the local RECO attempts to connect to remote databases and automatically complete the commit or rollback of the local portion of any pending distributed transactions.

Job Queue Processes Job queue processes are used for batch processing. Beginning with Oracle9*i*, Release 1 (9.0.1), job queue processes are managed dynamically. This enables job queue clients to use more job queue processes when required. The resources used by the new processes are released when they are idle.

> **See Also:**
>
> - *Oracle9i Database Administrator's Guide* for more information about job queues.
> - "Job Queue Processes" on page 9-12 for more information about dynamic job queue processes.

Dispatcher (D*nnn*) **Dispatchers** are optional background processes, present only when a multithreaded server configuration is used. At least one dispatcher process is created for every communication protocol in use (D000, . . ., D*nnn*). Each

dispatcher process is responsible for routing requests from connected user processes to available shared server processes and returning the responses back to the appropriate user processes.

Lock Manager Server (LMS) The Lock Manager Server process (LMS) is used for inter-instance locking in Oracle9*i* Real Application Clusters.

> **See Also:** "Oracle9i Real Application Clusters: Multiple Instance Systems" on page 1-4 for more information about the configuration of the lock process

Queue Monitor (QMN*n*) The **queue monitor(s)** are optional background processes that monitor the message queues for Oracle Advanced Queuing. You can configure up to 10 queue monitor processes.

Communications Software and Oracle Net Services

If the user and server processes are on different computers of a network or if user processes connect to shared server processes through dispatcher processes, the user process and server process communicate using Oracle Net Services. **Dispatchers** are optional background processes, present only in the multithreaded server configuration. **Oracle Net Services** is Oracle's interface to standard communications protocols that enables data transmission between computers.

> **See Also:** "Oracle and Oracle Net Services" on page 2-6

The Program Interface Mechanism

The **program interface** is the mechanism by which a user process communicates with a server process. It serves as a method of standard communication between any client tool or application (such as Oracle Forms) and Oracle software. Its functions are to:

- Act as a communications mechanism by formatting data requests, passing data, and trapping and returning errors
- Perform conversions and translations of data, particularly between different types of computers or to external user program datatypes

An Example of How Oracle Works

The following example illustrates an Oracle configuration where the user and associated server process are on separate machines (connected through a network):

1. An instance is currently running on the computer that is executing Oracle (often called the **host** or **database server**).
2. A computer running an application (a **local machine** or **client workstation**) runs the application in a user process. The client application attempts to establish a connection to the server using the proper Oracle Net Services driver.
3. The server is running the proper Oracle Net Services driver. The server detects the connection request from the application and creates a (dedicated) server process on behalf of the user process.
4. The user executes a SQL statement and commits the transaction. For example, the user changes a name in a row of a table.
5. The server process receives the statement and checks the shared pool for any shared SQL area that contains a similar SQL statement. If a shared SQL area is found, the server process checks the user's access privileges to the requested data, and the previously existing shared SQL area is used to process the statement. If not, a new shared SQL area is allocated for the statement so it can be parsed and processed.
6. The server process retrieves any necessary data values from the actual datafile (table) or those stored in the system global area.
7. The server process modifies data in the system global area. The DBW*n* process writes modified blocks permanently to disk when doing so is efficient. Because the transaction is committed, the LGWR process immediately records the transaction in the online redo log file.
8. If the transaction is successful, the server process sends a message across the network to the application. If it is not successful, an error message is transmitted.
9. Throughout this entire procedure, the other background processes run, watching for conditions that require intervention. In addition, the database server manages other users' transactions and prevents contention between transactions that request the same data.

These steps describe the most basic level of operations that Oracle performs.

> **See Also:** Chapter 9, "Process Architecture" for more information about Oracle configuration

Processes and Threads on Windows NT

In the UNIX environment, most Oracle processes are threads of one master Oracle process rather than being individual processes. On Windows NT, all processes consist of at least one **thread**. A thread is an individual execution within a process. Threads are objects within a process that execute program instructions. Threads enable concurrent operations within a process so that a process can execute different parts of its program simultaneously on different processors. A thread is the most fundamental component that can be scheduled on Windows NT.

In UNIX documentation such as this book, whenever the word "process" is mentioned, it is considered a "thread" on Windows NT.

> **See Also:** *Oracle9i Database Administrator's Guide for Windows* for more information about processes and threads on the Windows NT operating system

The Object-Relational Model for Database Management

Database management systems have evolved from hierarchical to network to relational models. The most widely accepted database model is the **relational model**. Oracle extends the relational model to an **object-relational model**, which makes it possible to store complex business models in a relational database.

The Relational Model

The relational model has three major characteristics:

Structures	Structures are well-defined objects (such as tables, views, indexes, and so on) that store or access the data of a database. Structures and the data contained within them can be manipulated by operations.
Operations	Operations are clearly defined actions that enable users to manipulate the data and structures of a database. The operations on a database must adhere to a predefined set of integrity rules.
Integrity rules	Integrity rules are the laws that govern which operations are allowed on the data and structures of a database. Integrity rules protect the data and the structures of a database.

Relational database management systems offer benefits such as:

- Independence of physical data storage and logical database structure
- Variable and easy access to all data
- Complete flexibility in database design
- Reduced data storage and redundancy

The Object-Relational Model

The object-relational model enables users to define **object types**, specifying both the structure of the data and the methods of operating on the data, and to use these datatypes within the relational model.

An object type differs from native SQL datatypes in that it is user-defined and it specifies both the underlying persistent data (attributes) and the related behaviors (methods).

Object types are abstractions of the real-world entities—for example, purchase orders—that application programs deal with. Object types are used to extend the modeling capabilities provided by the native datatypes.

An object type has three kinds of components:

- A **name**, which serves to identify the object type uniquely.
- **Attributes**, which are built-in datatypes or other user-defined types. Attributes model the structure of the real world entity.
- **Methods**, which are functions or procedures written in PL/SQL and stored in the database, or written in a language such as C or Java and stored externally. Methods implement specific operations that an application can perform on the data. Every object type has a **constructor method** that makes a new object according to the datatype's specification.

Type Inheritance

Oracle9*i*, Release 1 (9.0.1), supports inheritance of object types, including the ability to:

- Define, store, and retrieve subtype instances within the database.
- Manipulate them in languages, interfaces, utilities, and tools

Type inheritance is a way to organize types the same way types can be used for organizing objects. Inheritance provides a higher level of abstraction for managing complexity of the application model.

Type inheritance enables sharing of similarities between types as well as extending their characteristics. Sharing provides efficiency in development time and improves manageability of applications. Extensibility provides flexibility and power of expression to solving application problems.

Most object applications organize their objects into types, and types into type hierarchies. Empirically, it is sufficient to organize type hierarchies into sets of trees; thus, single type inheritance is sufficient to support type organization for most applications. A subtype may **extend** (inherit from) at most one supertype; furthermore, a supertype must be an object type and so a subtype is always an object type. A subtype inherits all its supertype's attributes and methods. A subtype may also add new (called **declared**) attributes and methods. All attributes, inherited and declared, and methods of a type are accessible to methods declared in the type and its subtype(s) or to other methods that have access to the type.

Type inheritance and **polymorphism** enable applications to use a shrink-wrapped type library as given, or to extend the type library by adding new attributes or methods to the library's types, or by **specializing** (**overriding**) already defined methods of the library's types. A subtype may override any method in its supertype chain.

The benefits of polymorphism derive partially from **substitutability**. Substitutability enables a value of some subtype to be used by code originally written for the supertype, without any specific knowledge of the subtype being needed in advance. The subtype value behaves to the surrounding code just like a value of the supertype would, even if it perhaps uses different mechanisms within its specializations of methods. **Instance substitutability** refers to the ability to use an object value of a subtype in a context declared in terms of a supertype; **REF substitutability** refers to the ability to use a REF to a subtype in a context declared in terms of a REF to a supertype.

Note: A REF is a reference to an object stored in a database table, instead of the object itself. REF types can occur in relational columns and also as datatypes of an object type.

See Also: *Oracle9i SQL Reference*

Description of Schemas and Schema Objects

A **schema** is a collection of database objects that are available to a user. **Schema objects** are the logical structures that directly refer to the database's data. Schema objects include such structures as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links. (There is no relationship between a tablespace and a schema. Objects in the same schema can be in different tablespaces, and a tablespace can hold objects from different schemas.)

Description of Tables

A **table** is the basic unit of data storage in an Oracle database. The tables of a database hold all of the user-accessible data.

Table data is stored in **rows** and **columns**. Every table is defined with a **table name** and set of columns. Each column is given a **column name**, a **datatype** (such as `CHAR`, `DATE`, or `NUMBER`), and a **width** (which could be predetermined by the datatype, as in `DATE`) or **scale** and **precision** (for the `NUMBER` datatype only). Once a table is created, valid rows of data can be inserted into it. The table's rows can then be queried, deleted, or updated.

To enforce defined business rules on a table's data, integrity constraints and triggers can also be defined for a table.

See Also:

- Chapter 18, "Triggers" for more information about defining triggers
- "Data Integrity" on page 2-17 for more information about business rules and integrity constraints

Description of Views

A **view** is a custom-tailored presentation of the data in one or more tables. A view can also be thought of as a "stored query."

Views do not actually contain or store data. Rather, they derive their data from the tables on which they are based, referred to as the **base tables** of the views. Base tables (also known as master tables) can in turn be tables or can themselves be views.

Like tables, views can be queried, updated, inserted into, and deleted from, with some restrictions. All operations performed on a view actually affect the base tables of the view.

Views are often used to do the following:

- Provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table. For example, a view of a table can be created so that columns with sensitive data (for example, salary information) are not included in the definition of the view.
- Hide data complexity. For example, a single view can combine 12 monthly sales tables to provide a year of data for analysis and reporting. A single view can also be used to create a **join**, which is a display of related columns or rows in multiple tables. However, the view hides the fact that this data actually originates from several tables.
- Present the data in a different perspective from that of the base table. For example, views provide a means to rename columns without affecting the tables on which the view is based.
- Store complex queries. For example, a query can perform extensive calculations with table information. By saving this query as a view, the calculations are performed only when the view is queried.

Views that involve a join (a `SELECT` statement that selects data from multiple tables) of two or more tables can only be updated under certain conditions.

> **See Also:** "Updatable Join Views" on page 11-20 for more information about updating join views

Materialized Views

A **materialized view** provides indirect access to table data by storing the results of a query in a separate schema object. Unlike an ordinary view, which does not take up any storage space or contain any data, a materialized view contains the rows resulting from a query against one or more base tables or views. A materialized view can be stored in the same database as its base table(s) or in a different database.

Materialized views stored in the same database as their base tables can improve query performance through **query rewrites**. Query rewrites are particularly useful in a data warehouse environment.

The optimizer can rewrite the query to access the precomputed results stored in a materialized view in the following cases:

- For queries that involve aggregate data or joins, if compatibility is set to Oracle8
- For queries that include filter selections, if compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher

See Also:

- *Oracle9i Data Warehousing Guide* for information about using materialized views in data warehousing
- "Table Replication" on page 2-5 for more information about replicating data in a remote database

Sequences

A **sequence** generates a serial list of unique numbers for numeric columns of a database's tables. Sequences simplify application programming by automatically generating unique numerical values for the rows of a single table or multiple tables.

For example, assume two users are simultaneously inserting new employee rows into the `EMP` table. By using a sequence to generate unique employee numbers for the `EMPNO` column, neither user has to wait for the other to enter the next available employee number. The sequence automatically generates the correct values for each user.

Sequence numbers are independent of tables, so the same sequence can be used for one or more tables. After creation, a sequence can be accessed by various users to generate actual sequence numbers.

Program Units

Program unit is used in this manual to refer to stored procedures, functions, packages, triggers, and anonymous blocks.

See Also: "Data Access Overview" on page 2-11 for more information about SQL and PL/SQL procedures, functions, and packages

Synonyms

A **synonym** is an alias for a table, view, sequence, or program unit. A synonym is not actually a schema object itself, but instead is a direct reference to a schema object. Synonyms are used to:

- Mask the real name and owner of a schema object
- Provide public access to a schema object

- Provide location transparency for tables, views, or program units of a remote database
- Simplify the SQL statements for database users

A synonym can be public or private. An individual user can create a **private synonym**, which is available only to that user. Database administrators most often create **public synonyms** that make the base schema object available for general, systemwide use by any database user.

Indexes

Indexes are optional structures associated with tables. Indexes can be created to increase the performance of data retrieval. Just as the index in this manual helps you locate specific information faster than if there were no index, an Oracle index provides a faster access path to table data.

When processing a request, Oracle can use some or all of the available indexes to locate the requested rows efficiently. Indexes are useful when applications often query a table for a range of rows (for example, all employees with a salary greater than 1000 dollars) or a specific row.

Indexes are created on one or more columns of a table. Once created, an index is automatically maintained and used by Oracle. Changes to table data (such as adding new rows, updating rows, or deleting rows) are automatically incorporated into all relevant indexes with complete transparency to the users.

Indexes are logically and physically independent of the data. They can be dropped and created any time with no effect on the tables or other indexes. If an index is dropped, all applications continue to function. However, access to previously indexed data could be slower.

You can **partition** indexes.

> **See Also:** Chapter 12, "Partitioned Tables and Indexes" for more information about partitioning indexes

Clusters and Hash Clusters

Clusters and hash clusters are optional structures for storing table data. They can be created to increase the performance of data retrieval.

Clustered Tables **Clusters** are groups of one or more tables physically stored together because they share common columns and are often used together. Because related rows are physically stored together, disk access time improves.

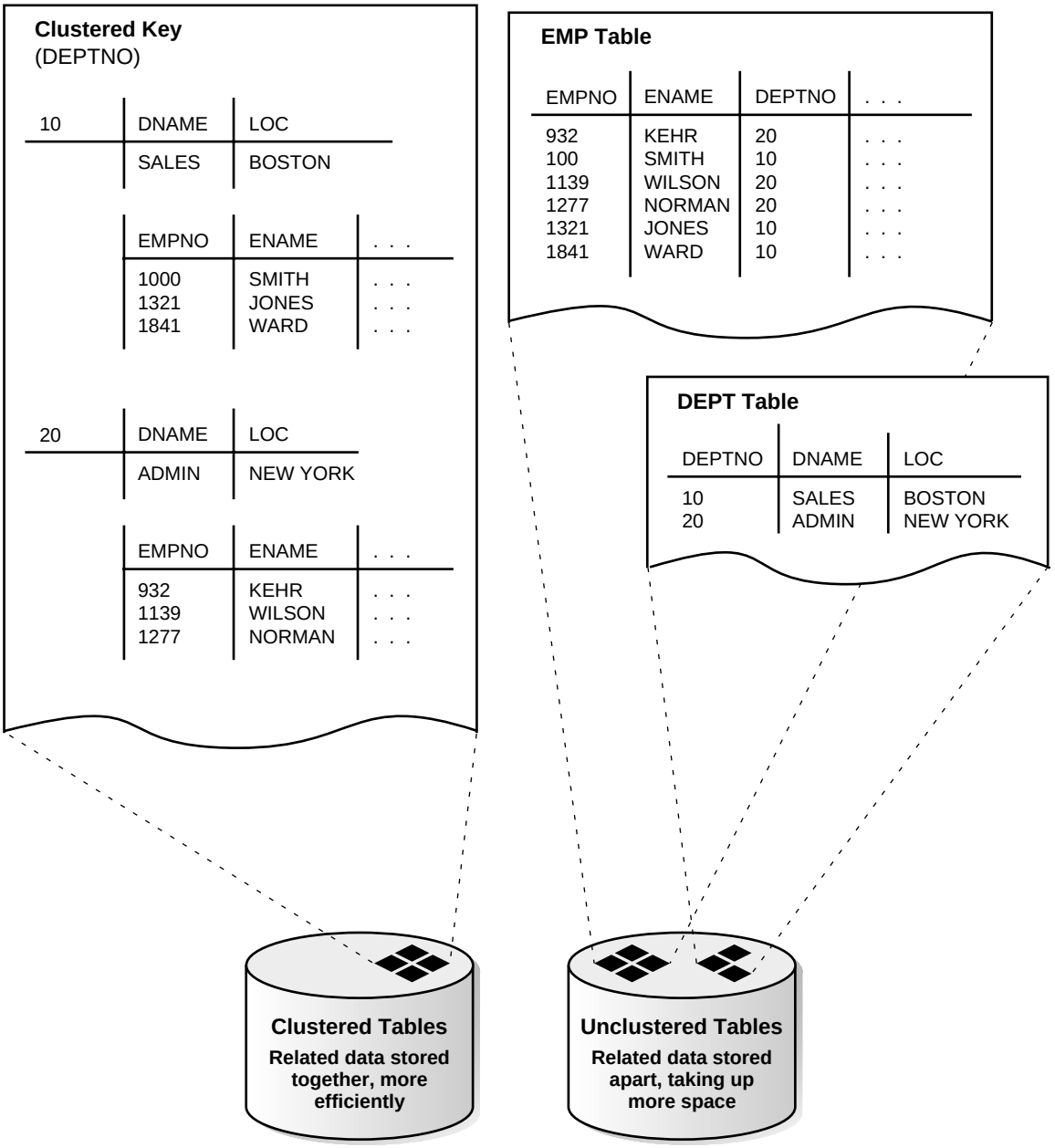
The related columns of the tables in a cluster are called the **cluster key**. The cluster key is indexed so that rows of the cluster can be retrieved with a minimum amount of I/O. Because the data in a cluster key of an index cluster (a nonhash cluster) is stored only once for multiple tables, clusters store a set of tables more efficiently than if the tables were stored individually (not clustered).

Clusters also can improve performance of data retrieval, depending on data distribution and what SQL operations are most often performed on the data. In particular, clustered tables that are queried in joins benefit from the use of clusters because the rows common to the joined tables are retrieved with the same I/O operation.

Like indexes, clusters do not affect application design. Whether or not a table is part of a cluster is transparent to users and to applications. Data stored in a clustered table is accessed by SQL in the same way as data stored in a nonclustered table.

Figure 1–3 illustrates how clustered and nonclustered data are physically stored.

Figure 1–3 Clustered and Nonclustered Tables



Hash Clusters **Hash clusters** cluster table data in a manner similar to normal, index clusters (clusters keyed with an index rather than a hash function). However, a row is stored in a hash cluster based on the result of applying a **hash function** to the row's cluster key value. All rows with the same key value are stored together on disk.

Hash clusters are a better choice than using an indexed table or index cluster when a table is often queried with equality queries (for example, return all rows for department 10). For such queries, the specified cluster key value is hashed. The resulting hash key value points directly to the area on disk that stores the rows.

Dimensions

A **dimension** defines hierarchical (parent/child) relationships between pairs of columns or column sets. Each value at the child level is associated with one and only one value at the parent level.

A dimension schema object is a container of logical relationships between tables and does not have any data storage assigned to it. 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 *level*s
- Optional `ATTRIBUTE` clauses, each of which identifies an additional column or column set associated with an individual `LEVEL`

The columns in a dimension can come either from the same table *(***denormalized***)* or from multiple tables *(***fully** or **partially normalized***)*. To define a dimension over columns from multiple tables, connect the tables by inner equijoins using the `JOIN` clause of the `HIERARCHY` clause.

> **See Also:** *Oracle9i Data Warehousing Guide* for information about using dimensions in materialized views and data warehousing

Database Links

A **database link** is a named schema object that describes a path from one database to another. Database links are implicitly used when a reference is made to a **global object name** in a distributed database.

> **See Also:** "Distributed Databases" on page 2-3

Data Dictionary

Each Oracle database has a **data dictionary**. An Oracle data dictionary is a set of tables and views that are used as a **read-only** reference about the database. For example, a data dictionary stores information about both the logical and physical structure of the database. In addition to this valuable information, a data dictionary also stores such information as:

- The valid users of an Oracle database
- Information about integrity constraints defined for tables in the database
- How much space is allocated for a schema object and how much of it is in use

A data dictionary is created when a database is created. To accurately reflect the status of the database at all times, the data dictionary is automatically updated by Oracle in response to specific actions (such as when the structure of the database is altered). The data dictionary is critical to the operation of the database, which relies on the data dictionary to record, verify, and conduct ongoing work. For example, during database operation, Oracle reads the data dictionary to verify that schema objects exist and that users have proper access to them.

See Also: Chapter 5, "The Data Dictionary"

Data Concurrency and Consistency Overview

This section explains the software mechanisms used by Oracle to fulfill the following important requirements of an information management system:

- Data must be read and modified in a consistent fashion.
- Data concurrency of a multiuser system must be maximized.
- High performance is required for maximum productivity from the many users of the database system.

Concurrency

A primary concern of a multiuser database management system is how to control **concurrency**, which is the simultaneous access of the same data by many users. Without adequate concurrency controls, data could be updated or changed improperly, compromising data integrity.

If many people are accessing the same data, one way of managing data concurrency is to make each user wait for a turn. The goal of a database management system is to reduce that wait so it is either nonexistent or negligible to each user. All data

manipulation language statements should proceed with as little interference as possible, and destructive interactions between concurrent transactions must be prevented. Destructive interaction is any interaction that incorrectly updates data or incorrectly alters underlying data structures. Neither performance nor data integrity can be sacrificed.

Oracle resolves such issues by using various types of locks and a multiversion consistency model. Both features are discussed later in this section. These features are based on the concept of a transaction. It is the application designer's responsibility to ensure that transactions fully exploit these concurrency and consistency features.

> **See Also:** "Data Consistency Using Transactions" on page 2-15 for more information about concurrency and consistency features

Read Consistency

Read consistency, as supported by Oracle, does the following:

- Guarantees that the set of data seen by a statement is consistent with respect to a single point in time and does not change during statement execution (statement-level read consistency)
- Ensures that readers of database data do not wait for writers or other readers of the same data
- Ensures that writers of database data do not wait for readers of the same data
- Ensures that writers only wait for other writers if they attempt to update identical rows in concurrent transactions

The simplest way to think of Oracle's implementation of read consistency is to imagine each user operating a private copy of the database, hence the multiversion consistency model.

Read Consistency, Undo Records, and Transactions

To manage the multiversion consistency model, Oracle must create a read-consistent set of data when a table is being queried (read) and simultaneously updated (written). When an update occurs, the original data values changed by the update are recorded in the database's undo records. As long as this update remains part of an uncommitted transaction, any user that later queries the modified data views the original data values. Oracle uses current information in the system global area and information in the undo records to construct a **read-consistent view** of a table's data for a query.

Only when a transaction is committed are the changes of the transaction made permanent. Statements that start *after* the user's transaction is committed only see the changes made by the committed transaction.

Note that a transaction is key to Oracle's strategy for providing read consistency. This unit of committed (or uncommitted) SQL statements:

- Dictates the start point for read-consistent views generated on behalf of readers
- Controls when modified data can be seen by other transactions of the database for reading or updating

Read-Only Transactions

By default, Oracle guarantees statement-level read consistency. The set of data returned by a single query is consistent with respect to a single point in time. However, in some situations, you may also require transaction-level read consistency. This is the ability to run multiple queries within a single transaction, all of which are read-consistent with respect to the same point in time, so that queries in this transaction do not see the effects of intervening committed transactions.

If you want to run a number of queries against multiple tables and if you are not doing any updating, you may prefer a **read-only transaction**. After indicating that your transaction is read-only, you can execute as many queries as you like against any table, knowing that the results of each query are consistent with respect to the same point in time.

Locking Mechanisms

Oracle also uses **locks** to control concurrent access to data. Locks are mechanisms intended to prevent destructive interaction between users accessing Oracle data.

Locks are used to achieve two important database goals:

Consistency	Ensures that the data a user is viewing or changing is not changed (by other users) until the user is finished with the data.
Integrity	Ensures that the database's data and structures reflect all changes made to them in the correct sequence.

Locks guarantee data integrity while enabling maximum concurrent access to the data by unlimited users.

Automatic Locking

Oracle locking is performed automatically and requires no user action. Implicit locking occurs for SQL statements as necessary, depending on the action requested.

Oracle's lock manager automatically locks table data at the row level. By locking table data at the row level, contention for the same data is minimized.

Oracle's lock manager maintains several different types of row locks, depending on what type of operation established the lock. In general, there are two types of locks: **exclusive locks** and **share locks**. Only one exclusive lock can be placed on a resource (such as a row or a table); however, many share locks can be placed on a single resource. Both exclusive and share locks always allow queries on the locked resource but prohibit other activity on the resource (such as updates and deletes).

Manual Locking

Under some circumstances, a user may want to override default locking. Oracle allows manual override of automatic locking features at both the row level (by first querying for the rows that will be updated in a subsequent statement) and the table level.

See Also: "Explicit (Manual) Data Locking" on page 22-31

Quiesce Database

Database administrators often need isolation from concurrent nonDBA actions, that is, isolation from concurrent nonDBA transactions, queries, or PL/SQL statements. One way to provide such isolation is to shut down the database and reopen it in restricted mode. The Quiesce Database feature introduced in Oracle9*i*, Release 1 (9.0.1), provides another way of providing isolation: to put the system into quiesced state without disrupting users.

The database administrator uses SQL statements to quiesce the database. Once the system is in quiesced state, the DBA can safely perform certain actions whose executions require isolation from concurrent nonDBA users.

See Also: "Quiesce Database" on page 22-15 for more information about quiescing a database

Startup and Shutdown Operations

An Oracle database is not available to users until the Oracle server has been started up and the database has been opened. These operations must be performed by the database administrator. Starting a database and making it available for systemwide use consists of three steps:

1. Start an instance of the Oracle server.
2. Mount the database.
3. Open the database.

When the Oracle server starts up, it uses a parameter file that contains initialization parameters. These parameters specify the name of the database, the amount of memory to allocate, the names of control files, and various limits and other system parameters.

Shutting down an instance and the database to which it is connected takes three steps:

1. Close the database.
2. Unmount the database.
3. Shut down the instance of the Oracle server.

Oracle automatically performs all three steps when an instance is shut down.

See Also:

- "Instance and Database Startup" on page 6-6 for more information about starting a database
- "Initialization Parameter Files" on page 6-4 for a sample initialization parameter file
- *Oracle9i Database Reference* for more information about initialization parameters
- "Database and Instance Shutdown" on page 6-11 for more information about shutting down an instance and the database to which it is connected

2

Oracle Server Features

This chapter provides an overview of some of the features of the Oracle server. The topics include:

- Internet Computing and Distributed Databases
- Advanced Queuing Overview
- Data Warehousing Overview
- Data Access Overview
- Database Security Overview
- Database Backup and Recovery Overview
- Introduction to Oracle9i Data Guard
- LogMiner SQL-Based Log Analyzer Overview

Note: This chapter contains information relating to both Oracle9*i* and the Oracle9*i* Enterprise Edition. Some of the features and options documented in this chapter are available only if you have purchased the Oracle9*i* Enterprise Edition. See *Oracle9i Database New Features* for information about the differences between Oracle9*i* and the Oracle9*i* Enterprise Edition.

Internet Computing and Distributed Databases

As internet computing becomes more prevalent in today's computing environments, database management systems must be able to take advantage of distributed processing and storage capabilities. This section explains the architectural features of Oracle that meet these requirements.

See Also:

- *Oracle Data Integration Overview*
- *Oracle9i Database Administrator's Guide* for more information about distributed databases

Client/Server Architecture: Distributed Processing

Distributed processing uses more than one processor to divide the processing for a set of related jobs. Distributed processing reduces the processing load on a single processor by allowing different processors to concentrate on a subset of related tasks, thus improving the performance and capabilities of the system as a whole.

An Oracle database system can easily take advantage of distributed processing by using its **client/server architecture**. In this architecture, the database system is divided into two parts: a front-end or a **client** and a back-end or a **server**.

The Client

The client is the front-end database application and is accessed by a user through the keyboard, display, and pointing device such as a mouse. The client has no data access responsibilities. It concentrates on requesting, processing, and presenting data managed by the server. The client workstation can be optimized for its job. For example, it may not need large disk capacity or it may benefit from graphic capabilities.

The Server

The server 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. The computer that manages the server can be optimized for its duties. For example, it can have large disk capacity and fast processors.

Multitier Architecture: Application Servers

A **multitier architecture** has the following components:

- A client or initiator process that starts an operation
- One or more application servers that perform parts of the operation. An **application server** provides access to the data for the client and performs some of the query processing, thus removing some of the load from the database server. It can serve as an interface between clients and multiple database servers, including providing an additional level of security.
- An end or database server that stores most of the data used in the operation

This architecture enables use of an application server to:

- Validate the credentials of a client, such as a web browser
- Connect to an Oracle database server
- Perform the requested operation on behalf of the client

The identity of the client is maintained throughout all tiers of the connection. The Oracle database server audits operations that the application server performs on behalf of the client separately from operations that the application server performs on its own behalf (such as a request for a connection to the database server). The application server's privileges are limited to prevent it from performing unneeded and unwanted operations during a client operation.

Distributed Databases

A **distributed database** is a network of databases managed by multiple database servers that appears to a user as a single logical database. The data of all databases in the distributed database can be simultaneously accessed and modified. The primary benefit of a distributed database is that the data of physically separate databases can be logically combined and potentially made accessible to all users on a network.

Each computer that manages a database in the distributed database is called a **node**. The database to which a user is directly connected is called the **local** database. Any additional databases accessed by this user are called **remote** databases. When a local database accesses a remote database for information, the local database is a client of the remote server. This is an example of client/server architecture.

While a distributed database enables increased access to a large amount of data across a network, it must also hide the location of the data and the complexity of accessing it across the network. The distributed database management system must

also preserve the advantages of administrating each local database as though it were not distributed.

Location Transparency

Location transparency occurs when the physical location of data is transparent to the applications and users of a database system. Several Oracle features, such as views, procedures, and synonyms, can provide location transparency. For example, a view that joins table data from several databases provides location transparency because the user of the view does not need to know from where the data originates.

Site Autonomy

Site autonomy means that each database participating in a distributed database is administered separately and independently from the other databases, as though each database were a non-networked database. Although each database can work with others, they are distinct, separate systems that are cared for individually.

Distributed Data Manipulation

The Oracle distributed database architecture supports all DML operations, including queries, inserts, updates, and deletes of remote table data. To access remote data, you make reference to the remote object's global object name. No coding or complex syntax is required to access remote data.

For example, to query a table named `EMP` in the remote database named `SALES`, reference the table's global object name:

```
SELECT * FROM emp@sales;
```

Two-Phase Commit

Oracle provides the same assurance of data consistency in a distributed environment as in a nondistributed environment. Oracle provides this assurance using the transaction model and a **two-phase commit mechanism**.

As in nondistributed systems, transactions should be carefully planned to include a logical set of SQL statements that should all succeed or fail as a unit. Oracle's two-phase commit mechanism guarantees that no matter what type of system or network failure occurs, a distributed transaction either commits on all involved nodes or rolls back on all involved nodes to maintain data consistency across the global distributed database.

See Also: "The Two-Phase Commit Mechanism" on page 17-10

Replication

Replication is the process of copying and maintaining database objects, such as tables, in multiple databases that make up a distributed database system. Changes applied at one site are captured and stored locally before being forwarded and applied at each of the remote locations. Oracle replication is a fully integrated feature of the Oracle server. It is not a separate server.

Replication uses distributed database technology to share data between multiple sites, but a replicated database and a distributed database are not the same. In a distributed database, data is available at many locations, but a particular table resides at only one location. For example, the `EMP` table can reside at only the `db1` database in a distributed database system that also includes the `db2` and `db3` databases. Replication means that the same data is available at multiple locations. For example, the `EMP` table may be available at `db1, db2,` and `db3.`

See Also: *Oracle9i Replication* for further information on this topic

Table Replication

Distributed database systems often locally replicate remote tables that are frequently queried by local users. By having copies of heavily accessed data on several nodes, the distributed database does not need to send information across a network repeatedly, thus helping to maximize the performance of the database application.

Data can be replicated using materialized views.

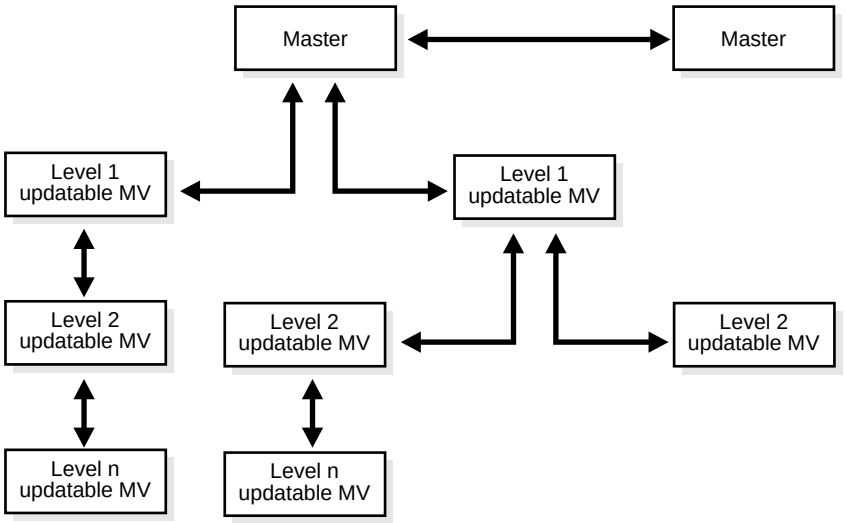
Multitier Materialized Views

Oracle9*i*, Release 1 (9.0.1), supports materialized views that are hierarchical and updatable. Multitier replication provides increased flexibility of design for a distributed application. Using multitier materialized views, applications can manage multilevel data subsets where there is no direct connection between levels.

An updatable materialized view lets you insert, update, and delete rows in the materialized view and propagate the changes to the target master table. Synchronous and asynchronous replication is supported.

Figure 2–1 shows an example of multitier architecture, diagrammed as an inverted tree structure. Changes are propagated up and down along the branches connecting the outermost materialized views with the master (the root).

Figure 2–1 Multitier Architecture



Conflict Resolution In Oracle9*i*, Release 1 (9.0.1), conflict resolution routines are defined at the topmost level, the master site, and are pulled into the updatable materialized view site when needed. This makes it possible to have multitier materialized views. Existing system-defined conflict resolution methods are supported.

In addition, users can write their own conflict resolution routines. A user-defined conflict resolution method is a PL/SQL function that returns either `TRUE` or `FALSE`. `TRUE` indicates that the method was able to successfully resolve all conflicting modifications for a column group.

> **See Also:** *Oracle9i Replication* and *Oracle9i SQL Reference* for more information about creating and managing multitier materialized views

Oracle and Oracle Net Services

Oracle Net Services is Oracle's mechanism for interfacing with the communication protocols used by the networks that facilitate distributed processing and distributed databases. Communication protocols define the way that data is transmitted and received on a network. In a networked environment, an Oracle database server communicates with client workstations and other Oracle database servers using Oracle Oracle Net Services software.

Oracle Net Services supports communications on all major network protocols, ranging from those supported by PC LANs to those used by the largest of mainframe computer systems.

Using Oracle Net Services, the application developer does not have to be concerned with supporting network communications in a database application. If a new protocol is used, the database administrator makes some minor changes, while the application requires no modifications and continues to function.

See Also: *Oracle9i Net Services Administrator's Guide*

Heterogeneous Services

Heterogeneous Services is a component within the Oracle9*i* database server that is necessary for accessing a non-Oracle database system.

The term "non-Oracle database system" refers to the following:

- Any system accessed by PL/SQL procedures written in C (that is, by external procedures)
- Any system accessed through SQL (that is, by Oracle Transparent Gateways and Generic Connectivity)
- Any system accessed procedurally (that is, by procedural gateways)

Heterogeneous Services makes it possible for Oracle9*i* database server users to do the following:

- Use Oracle SQL statements to retrieve data stored in non-Oracle systems.
- Use Oracle procedure calls to access non-Oracle systems, services, or application programming interfaces (APIs) from within an Oracle distributed environment.

Heterogeneous Services is generally applied in one of two ways:

- Users use an Oracle Transparent Gateway in conjunction with Heterogeneous Services to access a particular, vendor-specific, non-Oracle system for which an Oracle Transparent Gateways is designed. For example, you would use the Oracle Transparent Gateway for Sybase on Solaris to access a Sybase database system that was operating on a Sun Solaris platform.
- Users use Heterogeneous Services' generic connectivity to access non-Oracle databases through ODBC or OLE DB interfaces.

See Also: *Oracle9i Heterogeneous Connectivity Administrator's Guide* for more information

Advanced Queuing Overview

Oracle Advanced Queuing provides an infrastructure for distributed applications to communicate asynchronously using messages. Oracle Advanced Queuing stores messages in queues for deferred retrieval and processing by the Oracle server. This provides a reliable and efficient queuing system without additional software such as transaction processing monitors or message-oriented middleware.

Uses for Message Queuing

Consider a typical online sales business. It includes an order entry application, an order processing application, a billing application, and a customer service application. Physical shipping departments are located at various regional warehouses. The billing department and customer service department can also be located in different places.

This scenario requires communication between multiple clients in a distributed computing environment. Messages pass between clients and servers as well as between processes on different servers. An effective messaging system implements content-based routing, content-based subscription, and content-based querying.

A messaging system can be classified into one of two types:

- Synchronous Communication
- Asynchronous Communication

Synchronous Communication

Synchronous communication is based on the request/reply paradigm—a program sends a request to another program and waits until the reply arrives.

This model of communication (also called **online** or **connected**) is suitable for programs that need to get the reply before they can proceed with their work. Traditional client/server architectures are based on this model.

The major drawback of the synchronous model of communication is that the programs to whom the request is sent must be available and running for the calling application to work.

Asynchronous Communication

In the **disconnected** or **deferred** model, programs communicate asynchronously, placing requests in a queue and then proceeding with their work.

For example, an application might require entry of data or execution of an operation after specific conditions are met. The recipient program retrieves the request from the queue and acts on it. This model is suitable for applications that can continue with their work after placing a request in the queue — they are not blocked waiting for a reply.

For deferred execution to work correctly even in the presence of network, machine and application failures, the requests must be stored persistently, and processed *exactly once*. This can be achieved by combining persistent queuing with transaction protection.

Processing each client/server request exactly once is often important to preserve both the integrity and flow of a transaction. For example, if the request is an order for a number of shares of stock at a particular price, then execution of the request zero or two times is unacceptable even if a network or system failure occurs during transmission, receipt, or execution of the request.

Advantages of Oracle Advanced Queuing

Oracle Advanced Queuing supports the following:

- Point-to-point and publish-subscribe models of communication
- Structured payload that can be queried using SQL
- Recipient and subscription lists
- Persistent and nonpersistent queues
- Queue-level access control
- Asynchronous notification
- HTTP and SMTP access
- Nonrepudiation service

Using Oracle Advanced Queuing, you can:

- Propagate messages to other databases, local or remote.
- Remove multiple messages from the queue as a bundle, specify multiple recipients, and wait for messages on multiple queues.

- Prioritize messages and specify a window of execution for each message.
- Perform AQ operations through HTTP / SMTP, using XML formatting.
- Retain messages and message history for analysis.
- Keep statistics on messages stored in queues and propagated to other queues.
- Integrate transactions to simplify application development and management
- Use Oracle9*i* Real Application Clusters to achieve higher performance

Because Oracle Advanced Queuing queues are implemented in database tables, all the operational benefits of high availability, scalability, and reliability apply to queue data. In addition, database development and management tools can be used with queues.

Applications can access the queuing functionality through the native interface for Oracle Advanced Queuing (defined in PL / SQL, C / C++, Java, and Visual Basic) or through the Java Messaging Service interface for Oracle Advanced Queuing (a Java API based on the Java Messaging Service standard).

> **See Also:** *Oracle9i Application Developer's Guide - Advanced Queuing* for detailed information about Oracle Advanced Queuing

Data Warehousing Overview

Oracle9*i*, Release 1 (9.0.1), provides many features that support the needs of data warehousing and business intelligence. These features are typically useful for other activities such as Online Transaction Processing (OLTP). Therefore, most features described in this manual are presented in general terms, rather than emphasizing their data warehousing use.

These are some of the Oracle features that are designed to support data warehousing and business intelligence:

- Parallelism in queries, data loading, indexing, and other data manipulation
- Cost-based query optimization for efficient data access
- Indexing techniques specifically tailored to data warehousing needs
- Advanced SQL join algorithms
- Data partitioning to simplify management and enhance performance
- Analytic functions for high performance queries and simpler programming
- Materialized view for maximum performance

See Also: *Oracle9i Data Warehousing Guide* for details about using these features for data warehousing

Data Access Overview

This section introduces how Oracle meets the general requirements for a DBMS to:

- Adhere to industry accepted standards for a data access language
- Control and preserve the consistency of a database's information while manipulating its data
- Provide a system for defining and enforcing rules to maintain the integrity of a database's information
- Provide high performance

SQL—The Structured Query Language

SQL is a simple, powerful database access language that is the standard language for relational database management systems. For information about Oracle's compliance with ANSI/ISO standards, see *Oracle9i SQL Reference.*

SQL Statements

All operations on the information in an Oracle database are performed using SQL **statements**. A SQL statement is a string of SQL text that is given to Oracle to execute. A statement must be the equivalent of a complete SQL **sentence**, as in:

```
SELECT ename, deptno FROM emp;
```

Only a complete SQL statement can be executed, whereas a **sentence fragment**, such as the following, generates an error indicating that more text is required before a SQL statement can run:

```
SELECT ename
```

A SQL statement can be thought of as a very simple, but powerful, computer program or instruction. 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 statements
- Embedded SQL statements

Data Definition Language (DDL) Statements **Data definition language** statements define, maintain, and drop schema objects when they are no longer needed. DDL statements also include statements that permit a user to grant other users the **privileges**, or rights, to access the database and specific objects within the database.

> **See Also:** "Database Security Overview" on page 2-19 for more information about privileges

Data Manipulation Language (DML) Statements **Data manipulation language** statements manipulate the database's data. For example, querying, inserting, updating, and deleting rows of a table are all DML operations. Locking a table or view and examining the execution plan of an SQL statement are also DML operations.

Transaction Control Statements **Transaction control** statements manage the changes made by DML statements. They enable the user or application developer to group changes into logical transactions. Examples include `COMMIT`, `ROLLBACK`, and `SAVEPOINT`.

> **See Also:** "Transactions" on page 2-13 for more information about transaction control statements

Session Control Statements **Session control** statements let a user control the properties of his current session, including enabling and disabling roles and changing language settings. The two session control statements are `ALTER SESSION` and `SET ROLE`.

System Control Statements **System control** statements change the properties of the Oracle server instance. The only system control statement is `ALTER SYSTEM`. It lets you change such settings as the minimum number of shared servers, to kill a session, and to perform other tasks.

Embedded SQL Statements **Embedded SQL** statements incorporate DDL, DML, and transaction control statements in a procedural language program (such as those used with the Oracle precompilers). Examples include `OPEN`, `CLOSE`, `FETCH`, and `EXECUTE`.

Transactions

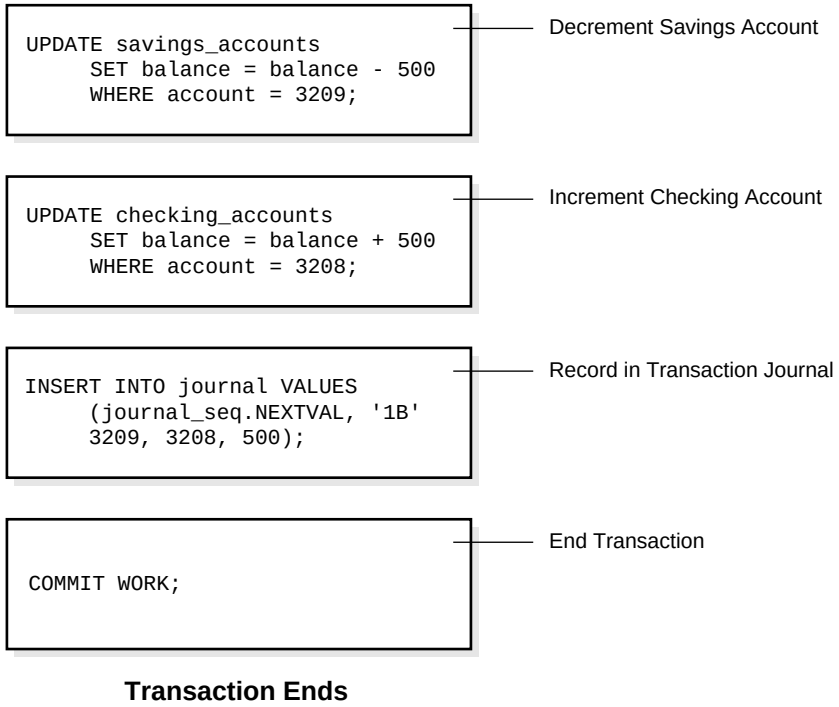
A **transaction** is a logical unit of work that comprises one or more SQL statements executed by a single user. According to the ANSI/ISO SQL standard, with which Oracle is compatible, a transaction begins with the user's first executable SQL statement. A transaction ends when it is explicitly committed or rolled back by that user.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations: decrease the savings account, increase the checking account, and record the transaction in the transaction journal.

Oracle must guarantee that all three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing (such as a hardware failure), the other statements of the transaction must be undone; this is called **rolling back**. If an error occurs in making any of the updates, then no updates are made.

Figure 2–2 illustrates the banking transaction example.

Figure 2–2 A Banking Transaction



Commit and Roll Back Transactions

The changes made by the SQL statements that constitute a transaction can be either committed or rolled back. After a transaction is committed or rolled back, the next transaction begins with the next SQL statement.

Committing a transaction makes permanent the changes resulting from all SQL statements in the transaction. The changes made by the SQL statements of a transaction become visible to other user sessions' transactions that start only after the transaction is committed.

Rolling back a transaction retracts any of the changes resulting from the SQL statements in the transaction. After a transaction is rolled back, the affected data is left unchanged as if the SQL statements in the transaction were never executed.

Savepoints

For long transactions that contain many SQL statements, intermediate markers, or **savepoints**, can be declared. Savepoints can be used to divide a transaction into smaller parts.

By using savepoints, you can arbitrarily mark your work at any point within a long transaction. This gives you the option of later rolling back all work performed from the current point in the transaction to a declared savepoint within the transaction. For example, you can use savepoints throughout a long complex series of updates, so if you make an error, you do not need to resubmit every statement.

Data Consistency Using Transactions

Transactions enable the database user or application developer to guarantee consistent changes to data, as long as the SQL statements within a transaction are grouped logically.

A transaction should consist of all of the necessary parts for one logical unit of work—no more and no less. Data in all referenced tables are in a consistent state before the transaction begins and after it ends. Transactions should consist of only the SQL statements that make one consistent change to the data.

For example, recall the banking example. A transfer of funds between two accounts (the transaction) should include increasing one account (one SQL statement), decreasing another account (one SQL statement), and the record in the transaction journal (one SQL statement). All actions should either fail or succeed together; the credit should not be committed without the debit. Other nonrelated actions, such as a new deposit to one account, should not be included in the transfer of funds transaction. Such statements should be in other transactions.

PL/SQL—Procedural Language Extension to SQL

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL combines the ease and flexibility of SQL with the procedural functionality of a structured programming language, such as `IF ... THEN`, `WHILE`, and `LOOP`.

When designing a database application, a developer should consider the advantages of using stored PL/SQL because:

- PL/SQL code can be stored centrally in a database. Network traffic between applications and the database is reduced, so application and system performance increases.

- Data access can be controlled by stored PL/SQL code. In this case, PL/SQL users can access data only as intended by the application developer (unless another access route is granted).
- PL/SQL blocks can be sent by an application to a database, executing complex operations without excessive network traffic.

Even when PL/SQL is not stored in the database, applications can send blocks of PL/SQL to the database rather than individual SQL statements, thereby again reducing network traffic.

The following sections describe the different program units that can be defined and stored centrally in a database.

Procedures and Functions

Procedures and **functions** consist of a set of SQL and PL/SQL statements that are grouped together as a unit to solve a specific problem or perform a set of related tasks. A procedure is created and stored in compiled form in the database and can be run by a user or a database application.

Procedures and functions are identical except that functions always return a single value to the caller, while procedures do not return values to the caller.

Packages Overview

Packages provide a method of encapsulating and storing related procedures, functions, variables, and other package constructs together as a unit in the database. Packages enable the administrator or application developer to organize such routines. They also offer increased functionality (for example, global package variables can be declared and used by any procedure in the package). They also improve performance (for example, all objects of the package are parsed, compiled, and loaded into memory once).

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i Supplied Java Packages Reference*

Database Triggers and Information Management

Oracle lets you write procedures written in PL/SQL, Java, or C that run implicitly whenever a table or view is modified or when some user actions or database system actions occur. These procedures are called **database triggers**.

Database triggers can be used in a variety of ways for the information management of your database. For example, they can be used to automate data generation, audit data modifications, enforce complex integrity constraints, and customize complex security authorizations.

Methods

A **method** is a procedure or function that is part of the definition of a user-defined datatype (object type, nested table, or variable array).

Methods are different from stored procedures in two ways:

- You invoke a method by referring to an object of its associated type.
- A method has complete access to the attributes of its associated object and to information about its type.

Every user-defined datatype has a system-defined **constructor method**; that is, a method that makes a new object according to the datatype's specification. The name of the constructor method is the name of the user-defined type. In the case of an object type, the constructor method's parameters have the names and types of the object type's attributes. The constructor method is a function that returns the new object as its value. Nested tables and arrays also have constructor methods.

Comparison methods define an order relationship among objects of a given object type. A **map method** uses Oracle's ability to compare built-in types. For example, Oracle can compare two rectangles by comparing their areas if an object type called `RECTANGLE` has attributes `HEIGHT` and `WIDTH` and you define a map method area that returns a number, namely the product of the rectangle's `HEIGHT` and `WIDTH` attributes. An **order method** uses its own internal logic to compare two objects of a given object type. It returns a value that encodes the order relationship. For example, it can return -1 if the first is smaller, 0 if they are equal, and 1 if the first is larger.

Data Integrity

It is very important to guarantee that data adheres to certain business rules, as determined by the database administrator or application developer. For example, assume that a business rule says that no row in the `INVENTORY` table can contain a numeric value greater than 9 in the `SALE_DISCOUNT` column. If an `INSERT` or `UPDATE` statement attempts to violate this integrity rule, Oracle must roll back the invalid statement and return an error to the application. Oracle provides integrity constraints and database triggers to manage a database's data integrity rules.

Integrity Constraints

An **integrity constraint** is a declarative way to define a business rule for a column of a table. An integrity constraint is a statement about a table's data that is always true and that follows these rules:

- If an integrity constraint is created for a table and some existing table data does not satisfy the constraint, the constraint cannot be enforced.
- After a constraint is defined, if any of the results of a DML statement violate the integrity constraint, the statement is rolled back and an error is returned.

Integrity constraints are defined with a table and are stored as part of the table's definition, centrally in the database's data dictionary, so that all database applications must adhere to the same set of rules. If a rule changes, it need only be changed once at the database level and not many times for each application.

The following integrity constraints are supported by Oracle:

NOT NULL	Disallows nulls (empty entries) in a table's column.
UNIQUE	Disallows duplicate values in a column or set of columns.
PRIMARY KEY	Disallows duplicate values and nulls in a column or set of columns.
FOREIGN KEY	Requires each value in a column or set of columns to match a value in a related table's UNIQUE or PRIMARY KEY. FOREIGN KEY integrity constraints also define referential integrity actions that dictate what Oracle should do with dependent data if the data it references is altered.
CHECK	Disallows values that do not satisfy the logical expression of the constraint.

Keys

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 different tables and columns of a relational database. The different types of keys include:

Primary key	The column or set of columns included in the definition of a table's PRIMARY KEY constraint. A primary key's values uniquely identify the rows in a table. Only one primary key can be defined for each table.
Unique key	The column or set of columns included in the definition of a UNIQUE constraint.
Foreign key	The column or set of columns included in the definition of a referential integrity constraint.
Referenced key	The unique key or primary key of the same or different table that is referenced by a foreign key.

Individual values in a key are called **key values**.

Database Triggers and Data Integrity

Database triggers let you define and enforce integrity rules, but a database trigger is not the same as an integrity constraint. Among other things, a database trigger defined to enforce an integrity rule does not check data already loaded into a table. Therefore, it is strongly recommended that you use database triggers only when the integrity rule cannot be enforced by integrity constraints.

Database Security Overview

Multiuser database systems, such as Oracle, include security features that control how a database is accessed and used. For example, security mechanisms:

- Prevent unauthorized database access
- Prevent unauthorized access to schema objects
- Control disk usage
- Control system resource use (such as CPU time)
- Audit user actions

Associated with each database user is a schema by the same name. By default, each database user creates and has access to all objects in the corresponding schema.

Database security can be classified into two categories: **system security** and **data security**.

System security includes the mechanisms that control the access and use of the database at the system level. For example, system security includes:

- Valid username / password combinations
- The amount of disk space available to a user's schema objects
- The resource limits for a user

System security mechanisms check whether a user is authorized to connect to the database, whether database auditing is active, and which system operations a user can perform.

Data security includes the mechanisms that control the access and use of the database at the schema object level. For example, data security includes:

- Which users have access to a specific schema object and the specific types of actions allowed for each user on the schema object (for example, user `SCOTT` can issue `SELECT` and `INSERT` statements but not `DELETE` statements using the `EMP` table)
- The actions, if any, that are audited for each schema object

Security Mechanisms

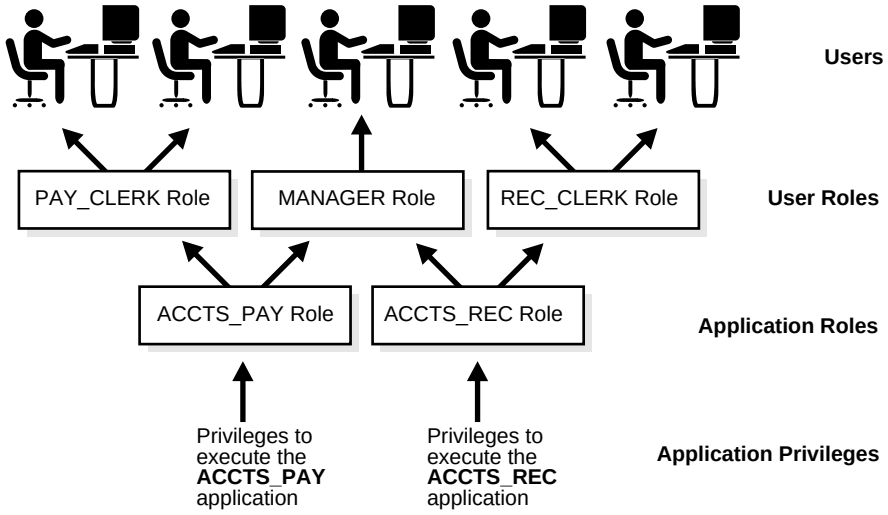
The Oracle server provides **discretionary access control**, which is a means of restricting access to information based on privileges. The appropriate privilege must be assigned to a user in order for that user to access a schema object. Appropriately privileged users can grant other users privileges at their discretion. For this reason, this type of security is called **discretionary**.

Oracle manages database security using several different facilities:

- Database users and schemas
- Privileges
- Roles
- Storage settings and quotas
- Resource limits
- Auditing

Figure 2–3 illustrates the relationships of the different Oracle security facilities, and the following sections provide an overview of users, privileges, and roles.

Figure 2–3 Oracle Security Features



Database Users and Schemas

Each Oracle database has a list of usernames. To access a database, a user must use a database application and attempt a connection with a valid username of the database. Each username has an associated password to prevent unauthorized use.

Security Domain Each user has a **security domain**—a set of properties that determine such things as:

- The actions (privileges and roles) available to the user
- The tablespace quotas (available disk space) for the user
- The system resource limits (for example, CPU processing time) for the user

Each property that contributes to a user's security domain is discussed in the following sections.

Privileges

A **privilege** is a right to execute a particular type of SQL statement. Some examples of privileges include the right to:

- Connect to the database (create a session)

- Create a table in your schema
- Select rows from someone else's table
- Execute someone else's stored procedure

The privileges of an Oracle database can be divided into two categories: **system privileges** and **schema object privileges**.

System Privileges **System privileges** allow users to perform a particular systemwide action or a particular action on a particular type of schema object. For example, the privileges to create a tablespace or to delete the rows of any table in the database are system privileges. Many system privileges are available only to administrators and application developers because the privileges are very powerful.

Schema Object Privileges **Schema object privileges** allow users to perform a particular action on a specific schema object. For example, the privilege to delete rows of a specific table is an object privilege. Object privileges are granted (assigned) to users so that they can use a database application to accomplish specific tasks.

Granted Privileges Privileges are granted to users so that users can access and modify data in the database. A user can receive a privilege two different ways:

- Privileges can be granted to users explicitly. For example, the privilege to insert records into the `EMP` table can be explicitly granted to the user `SCOTT`.
- Privileges can be granted to **roles** (a named group of privileges), and then the role can be granted to one or more users. For example, the privilege to insert records into the `EMP` table can be granted to the role named `CLERK`, which in turn can be granted to the users `SCOTT` and `BRIAN`.

Because roles enable easier and better management of privileges, privileges are normally granted to roles and not to specific users. The following section explains more about roles and their use.

Roles

Oracle provides for easy and controlled privilege management through roles. **Roles** are named groups of related privileges that you grant to users or other roles.

> **See Also:** "Introduction to Roles" on page 25-17 for more information about role properties

Storage Settings and Quotas

Oracle provides a way to direct and limit the use of disk space allocated to the database for each user, including default and temporary tablespaces and tablespace quotas.

Default Tablespace Each user is associated with a **default tablespace**. When a user creates a table, index, or cluster and no tablespace is specified to physically contain the schema object, the user's default tablespace is used if the user has the privilege to create the schema object and a quota in the specified default tablespace. The default tablespace feature provides Oracle with information to direct space usage in situations where schema object's location is not specified.

Temporary Tablespace Each user has a **temporary tablespace**. When a user executes a SQL statement that requires the creation of temporary segments (such as the creation of an index), the user's temporary tablespace is used. By directing all users' temporary segments to a separate tablespace, the temporary tablespace feature can reduce I/O contention among temporary segments and other types of segments.

Tablespace Quotas Oracle can limit the collective amount of disk space available to the objects in a schema. **Quotas** (space limits) can be set for each tablespace available to a user. The tablespace quota security feature permits selective control over the amount of disk space that can be consumed by the objects of specific schemas.

Profiles and Resource Limits

Each user is assigned a **profile** that specifies limitations on several system resources available to the user, including the following:

- Number of concurrent sessions the user can establish
- CPU processing time available for:
  - The user's session
  - A single call to Oracle made by a SQL statement
- Amount of logical I/O available for:
  - The user's session
  - A single call to Oracle made by a SQL statement
- Amount of idle time available for the user's session
- Amount of connect time available for the user's session

- Password restrictions:
  - Account locking after multiple unsuccessful login attempts
  - Password expiration and grace period
  - Password reuse and complexity restrictions

Different profiles can be created and assigned individually to each user of the database. A default profile is present for all users not explicitly assigned a profile. The resource limit feature prevents excessive consumption of global database system resources.

Selective Auditing of User Actions

Oracle permits selective **auditing** (recorded monitoring) of user actions to aid in the investigation of suspicious database use. Auditing can be performed at three different levels: statement auditing, privilege auditing, and schema object auditing.

Statement auditing

Statement auditing is the auditing of specific SQL statements without regard to specifically named schema objects. In addition, database triggers enable a DBA to extend and customize Oracle's built-in auditing features.

Statement auditing can be broad and audit all users of the system or can be focused to audit only selected users of the system. For example, statement auditing by user can audit connections to and disconnections from the database by the users `SCOTT` and `LORI`.

Privilege auditing

Privilege auditing is the auditing of powerful system privileges without regard to specifically named schema objects. Privilege auditing can be broad and audit all users or can be focused to audit only selected users.

Schema object auditing

Schema object auditing is the auditing of access to specific schema objects without regard to user. Object auditing monitors the statements permitted by object privileges, such as `SELECT` or `DELETE` statements on a given table.

For all types of auditing, Oracle allows the selective auditing of successful statement executions, unsuccessful statement executions, or both. This enables

monitoring of suspicious statements, regardless of whether the user issuing a statement has the appropriate privileges to issue the statement. The results of audited operations are recorded in a table called the **audit trail**. Predefined views of the audit trail are available so you can easily retrieve audit records.

Database Backup and Recovery Overview

This section covers the structures and software mechanisms used by Oracle to provide:

- Database recovery required by different types of failures
- Flexible recovery operations to suit any situation
- Availability of data during backup and recovery operations so users of the system can continue to work

Why Recovery Is Important

In every database system, the possibility of a system or hardware failure always exists. If a failure occurs and affects the database, the database must be recovered. The goals after a failure are to ensure that the effects of all committed transactions are reflected in the recovered database and to return to normal operation as quickly as possible while insulating users from problems caused by the failure.

Types of Failures

Several circumstances can halt the operation of an Oracle database. The most common types of failure are described as follows:

User error — User errors can require a database to be recovered to a point in time before the error occurred. For example, a user may accidentally drop a table. To enable recovery from user errors and accommodate other unique recovery requirements, Oracle provides exact point-in-time recovery. For example, if a user accidentally drops a table, the database can be recovered to the instant in time before the table was dropped.

Statement failure and **process failure**

Statement failure occurs when there is a logical failure in the handling of a statement in an Oracle program (for example, the statement is not a valid SQL construction). When statement failure occurs, the effects (if any) of the statement are automatically undone by Oracle and control is returned to the user.

A **process failure** is a failure in a user process accessing Oracle, such as an abnormal disconnection or process termination. The failed user process cannot continue work, although Oracle and other user processes can. The Oracle background process PMON automatically detects the failed user process or is informed of it by SQL*Net. PMON resolves the problem by rolling back the uncommitted transaction of the user process and releasing any resources that the process was using.

Common problems such as erroneous SQL statement constructions and aborted user processes should never halt the database system as a whole. Furthermore, Oracle automatically performs necessary recovery from uncommitted transaction changes and locked resources with minimal impact on the system or other users.

Instance failure

Instance failure occurs when a problem arises that prevents an instance from continuing work. Instance failure can result from a hardware problem such as a power outage, or a software problem such as an operating system crash. When an instance failure occurs, the data in the buffers of the system global area is not written to the datafiles.

Instance failure requires **crash recovery** or **instance recovery**. Crash recovery is automatically performed by Oracle when the instance restarts. In an Oracle9*i* Real Application Clusters environment, the SMON process of another instance performs instance recovery. The redo log is used to recover the committed data in the SGA's database buffers that was lost due to the instance failure.

Media (disk) failure An error can occur when trying to write or read a file that is required to operate the database. This is called disk failure because there is a physical problem reading or writing physical files on disk. A common example is a disk head crash, which causes the loss of all files on a disk drive.

Different files can be affected by this type of disk failure, including the datafiles, the redo log files, and the control files. Also, because the database instance cannot continue to function properly, the data in the database buffers of the system global area cannot be permanently written to the datafiles.

A disk failure requires **media recovery**. Media recovery restores a database's datafiles so the information in them corresponds to the most recent time point before the disk failure, including the committed data in memory that was lost because of the failure. To complete a recovery from a disk failure, the following is required: backups of the database's datafiles, and all online and necessary archived redo log files.

Oracle provides for complete recovery from all possible types of hardware failures, including disk crashes. Options are provided so that a database can be completely recovered or partially recovered to a specific point in time.

If some datafiles are damaged in a disk failure but most of the database is intact and operational, the database can remain open while the required tablespaces are individually recovered. Therefore, undamaged portions of a database are available for normal use while damaged portions are being recovered.

Structures Used for Recovery

Oracle uses several structures to provide complete recovery from an instance or disk failure: the **redo log**, **undo records**, a **control file**, and **database backups**. If compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher, undo records can be stored in either rollback segments or undo tablespaces.

> **See Also:** "Automatic Undo Management" on page 3-22 for more information about managing undo records

The Redo Log

The **redo log** is a set of files that protect altered database data in memory that has not been written to the datafiles. The redo log can consist of two parts: the **online redo log** and the **archived redo log**.

The Online Redo Log The **online redo log** is a set of two or more **online redo log files** that record all changes made to the database, including both uncommitted and committed changes. Redo entries are temporarily stored in redo log buffers of the system global area, and the background process LGWR writes the redo entries sequentially to an online redo log file. LGWR writes redo entries continually, and it also writes a commit record every time a user process commits a transaction.

The online redo log files are used in a cyclical fashion. For example, if two files constitute the online redo log, the first file is filled, the second file is filled, the first file is reused and filled, the second file is reused and filled, and so on. Each time a file is filled, it is assigned a *log sequence number* to identify the set of redo entries.

To avoid losing the database due to a single point of failure, Oracle can maintain multiple sets of online redo log files. A **multiplexed online redo log** consists of copies of online redo log files physically located on separate disks. Changes made to one member of the group are made to all members.

If a disk that contains an online redo log file fails, other copies are still intact and available to Oracle. System operation is not interrupted, and the lost online redo log files can be easily recovered using an intact copy.

The Archived Redo Log Optionally, filled online redo files can be archived before being reused, creating an **archived redo log**. Archived (offline) redo log files constitute the archived redo log.

The presence or absence of an archived redo log is determined by the mode that the redo log is using:

ARCHIVELOG	The filled online redo log files are archived before they are reused in the cycle.
NOARCHIVELOG	The filled online redo log files are not archived.

In ARCHIVELOG mode, the database can be completely recovered from both instance and disk failure. The database can also be backed up while it is open and

available for use. However, additional administrative operations are required to maintain the archived redo log.

If the database's redo log operates in NOARCHIVELOG mode, the database can be completely recovered from instance failure but not from disk failure. Also, the database can be backed up only while it is completely closed. Because no archived redo log is created, no extra work is required by the database administrator.

Undo Records

If compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher, undo records can be stored in either rollback segments or undo tablespaces.

Rollback Segment Undo Rollback segments have traditionally stored undo information used by several functions of Oracle. During database recovery, after all changes recorded in the redo log have been applied, Oracle uses rollback segment information to undo any uncommitted transactions. Because rollback segments are stored in the database buffers, this important recovery information is automatically protected by the redo log.

Automatic Undo Management Oracle9*i*, Release 1 (9.0.1), offers another method of storing undo records: using undo tablespaces. Automatic undo management has the following advantages over the traditional rollback segments:

- Automatic undo management eliminates the complexities of managing rollback segment space.
- Automatic undo management enables DBAs to exert control over how long undo records are retained before being overwritten.

Oracle recommends operating in automatic undo management mode. The database server can manage undo more efficiently, and automatic undo management mode is less complex to implement and manage.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about managing undo space
- "Undo Space Acquisition and Management" on page 6-9 for information about specifying the undo method at startup
- "The Rollback Segment SYSTEM" on page 3-31 and "Automatic Undo Management" on page 3-22 for more information about managing undo space

Control Files

The **control files** of a database keep, among other things, information about the file structure of the database and the current log sequence number being written by LGWR. During normal recovery procedures, the information in a control file is used to guide the automated progression of the recovery operation.

Multiplexed Control Files Oracle can maintain a number of identical control files, updating all of them simultaneously.

Database Backups

Because one or more files can be physically damaged as the result of a disk failure, media recovery requires the restoration of the damaged files from the most recent operating system backup of a database. There are several ways to back up the files of a database.

Whole Database Backups A whole database backup is an operating system backup of all datafiles, online redo log files, and the control file of an Oracle database. A whole database backup is performed when the database is closed and unavailable for use.

Partial Backups A partial backup is an operating system backup of part of a database. The backup of an individual tablespace's datafiles or the backup of a control file are examples of partial backups. Partial backups are useful only when the database's redo log is operated in ARCHIVELOG mode.

A variety of partial backups can be taken to accommodate any backup strategy. For example, you can back up datafiles and control files when the database is open or closed, or when a specific tablespace is online or offline. Because the redo log is operated in ARCHIVELOG mode, additional backups of the redo log are not necessary. The archived redo log is a backup of filled online redo log files.

Basic Recovery Steps

Because of the way DBW*n* writes database buffers to datafiles, at any given time a datafile might contain some tentative modifications by uncommitted transactions and might not contain some modifications by committed transactions. Therefore, two potential situations can result after a failure:

- Data blocks containing committed modifications were not written to the datafiles, so the changes appear only in the redo log. Therefore, the redo log contains committed data that must be applied to the datafiles.

- Because the redo log can contain data that was not committed, uncommitted transaction changes applied by the redo log during recovery must be erased from the datafiles.

To solve this situation, two separate steps are always used by Oracle during recovery from an instance or media failure: rolling forward and rolling back.

Rolling Forward

The first step of recovery is to **roll forward**, which is to reapply to the datafiles all of the changes recorded in the redo log. Rolling forward proceeds through as many redo log files as necessary to bring the datafiles forward to the required time.

If all necessary redo information is online, Oracle rolls forward automatically when the database starts. After roll forward, the datafiles contain all committed changes as well as any uncommitted changes that were recorded in the redo log.

Rolling Back

The roll forward is only half of recovery. After the roll forward, any changes that were not committed must be undone. After the redo log files have been applied, then the undo records are used to identify and undo transactions that were never committed yet were recorded in the redo log. This process is called **rolling back**. Oracle completes this step automatically.

See Also: *Oracle9i Backup and Recovery Concepts*

Transparent Application Failover (TAF)

Rapid recovery minimizes the time data is unavailable to users, but it does not address the disruption caused when user sessions fail. Users need to reestablish connections to the database, and work in progress can be lost. Transparent Application Failover (TAF) can mask many failures from users, restoring any session state and resuming queries that had been in progress at the time of the failure. Developers can further extend these capabilities by building applications that leverage TAF and make all failures, including those affecting transactions, transparent to users.

See Also: *Oracle9i Net Services Administrator's Guide* for more information about TAF

Recovery Manager

Recovery Manager (RMAN) is an Oracle utility that manages backup and recovery operations, creates backups of database files (datafiles, control files, and archived redo log files), and restores or recovers a database from backups.

Recovery Manager maintains a repository called the **recovery catalog**, which contains information about backup files and archived log files. Recovery Manager uses the recovery catalog to automate both restore operations and media recovery.

The recovery catalog contains:

- Information about backups of datafiles and archive logs
- Information about datafile copies
- Information about archived redo logs and copies of them
- Information about the physical schema of the target database
- Named sequences of statements called **stored scripts**

> **See Also:** *Oracle9i Recovery Manager User's Guide* for more information about Recovery Manager

Instance Recovery Tuning

There are several methods for improving performance of instance and crash recovery to keep the duration of recovery within user-specified bounds. For example:

- Use initialization parameters to influence the number of redo log records and data blocks involved in recovery.
- Size the redo log file to influence checkpointing frequency.
- Use SQL statements to initiate checkpoints.
- Parallelize instance recovery operations to further shorten the recovery time.

Besides using checkpoints to tune instance recovery, you can also use a variety of parameters to control Oracle's behavior during the rolling forward and rolling back phases of instance recovery. In some cases, you can parallelize operations and thereby increase recovery efficiency.

How Redo Log Size Influences Checkpointing Frequency

The size of a redo log file directly influences checkpoint performance. The smaller the size of the smallest log, the more aggressively Oracle writes dirty buffers to disk to ensure that the position of the checkpoint has advanced to the current log before that log completely fills. Forcing the checkpoint to advance into the current log before it fills means that Oracle does not need to wait for the checkpoint to advance out of a redo log file before it can be reused. Oracle enforces this behavior by ensuring that the number of redo blocks between the checkpoint and the most recent redo record is less than 90% of the size of the smallest log.

If the redo logs are small compared to the number of changes made against the database, then Oracle must switch logs frequently. If the value of `LOG_CHECKPOINT_INTERVAL` is less than 90% of the size of the smallest log, then the size of the smallest log file does not influence checkpointing behavior.

Although you specify the number and sizes of online redo log files at database creation, you can alter the characteristics of the redo log files after startup. Use the `ADD LOGFILE` clause of the `ALTER DATABASE` statement to add a redo log file and specify its size, or the `DROP LOGFILE` clause to drop a redo log.

The size of the redo log appears in the `LOG_FILE_SIZE_REDO_BLKS` column of the `V$INSTANCE_RECOVERY` dynamic performance view. This value shows how the size of the smallest online redo log is affecting checkpointing. By increasing or decreasing the size of the online redo logs, you indirectly influence the frequency of checkpoint writes.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for more information about tuning instance recovery

Use of SQL Statements to Initiate Checkpoints

Besides setting initialization parameters and sizing your redo log files, you can also influence checkpoints with SQL statements. `ALTER SYSTEM CHECKPOINT` directs Oracle to record a checkpoint for the node, and `ALTER SYSTEM CHECKPOINT GLOBAL` directs Oracle to record a checkpoint for every node in a cluster.

SQL-induced checkpoints are **heavyweight**. This means that Oracle records the checkpoint in a control file shared by all the redo threads. Oracle also updates the datafile headers. SQL-induced checkpoints move the checkpoint position to the point that corresponded to the end of the log when the statement was initiated. These checkpoints can adversely affect performance, because the additional writes to the datafiles increase system overhead.

See Also: *Oracle9i SQL Reference* for more information about these SQL statements

Introduction to Oracle9*i* Data Guard

Oracle's Standby Database is the most frequently used and, for most environments, the most effective disaster recovery solution for Oracle databases. Oracle9*i*, Release 1 (9.0.1), provides enhancements that do much more than meet essential disaster recovery requirements. By automating complex tasks and providing dramatically enhanced monitoring, alert, and control mechanisms, Standby Database and a number of new modules now help you to survive mistakes, corruptions and other disasters that might otherwise destroy your database. Also, the downtime required for upgrades, such as hardware and operating system maintenance, can be significantly reduced using Oracle9*i* standby databases.

As well as enhancing Standby Database, several entirely new components have been added that provide further protection against user errors and corruptions. These new components, together with the enhanced Standby Database, are contained in a new Oracle9*i* database feature called Oracle9*i* Data Guard.

See Also: *Oracle9i Data Guard Concepts and Administration*

Background of Oracle9*i* Data Guard

Oracle8 Automated Standby Database

The Oracle8 Automated Standby Database feature provides the means to create and automatically maintain copies of a production database to protect against disasters.

Oracle8 Automated Standby Database configuration performs the following functions:

- Updates the standby databases by automatically shipping archive logs as the primary database creates them.
- Includes a production (primary) database linked to one to four standby databases that are identical copies of the production database. Each can be used to take over production processing from the production database.
- Applies the archived redo logs to each standby using standard Oracle recovery. Logs are applied automatically, or they can accumulate for manual updating.
- The primary database is open and active. The standby database is either in recovery or open read-only.

Figure 2–4 Oracle8 Automated Standby Database

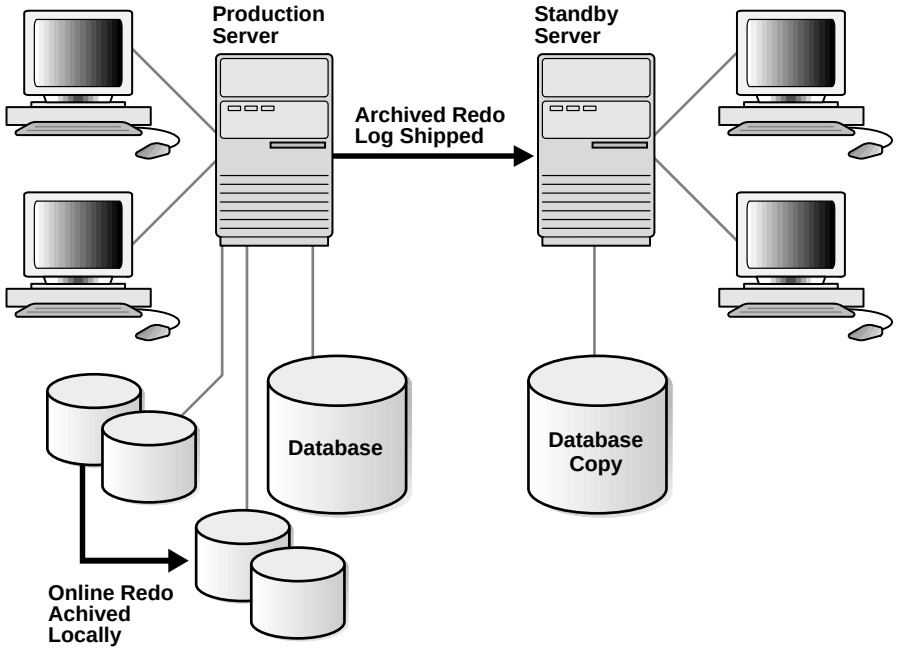


Figure 2–4 shows that as the online redo logs are archived locally, they are concurrently shipped to the standby database(s) through Oracle Net Services.

Most Oracle8-based disaster protection solutions include Automated Standby Database. Any application can use Automated Standby Database, because any Oracle database can be rebuilt with a backup and its logs using standard Oracle recovery. The performance impact of log shipping on primary-side performance is negligible.

What Kind of High Availability: Fast Failover or Disaster Protection?

Cluster technologies that are used for fast failover are often referred to as high availability solutions. But maintaining enterprise application continuity after a disaster or corruption is also a critical part of achieving high service levels. Fast failover cluster technologies, and the problems they solve, are quite different from those used for disaster protection. While these two areas are not mutually exclusive, from a practical perspective, the technologies used in one area often are not applied to solve problems in the other.

Fast Failover Cluster-based high availability solutions include more than one system or instance pointed at the same pool of disks. A high availability cluster can include multiple active and independent nodes working together on one database, or just one node that operates on the database at a time and then fails over to another single node on the same cluster. Fast failover solutions minimize downtime by detecting and identifying system faults or failures, and then failing over quickly. Any disaster that destroys or corrupts data does so for all potential failover target systems. Even with a fully hardware-mirrored disk configuration, any corruption written to one disk is immediately propagated to the mirror; there is no practical way to prevent this by checking at the hardware level.

Scalability is an additional advantage of multisystem clusters that operate on one database. Oracle9*i* Real Application Clusters is an example of this kind of a cluster solution that provides both scalability as well as enterprise-class availability.

Disaster Protection There are no shared disks in typical disaster protection, or in DR (disaster recovery) configurations. The first priority is to ensure that copies of the data are elsewhere, usable, and correct when any data-destroying disaster strikes. Each site's peculiar risk profile determines the likelihood of disasters like floods, hurricanes, and earthquakes, and therefore the distance needed to ensure that one site's risks are independent of the other. That is, the standby database must be far enough away from the primary system's location so that the same events likely to disable the primary are unlikely to also cripple the standby. In hurricane or earthquake country, this can mean more than 250 miles. Hardware and cluster-based mirroring technologies currently provide solutions that either do not scale to meet these needs, or do so in configurations that are difficult to economically justify.

You can implement Automated Standby Database locally, that is, connected to the primary using a local area network (LAN). The standby database would be a failover option despite slower failover time compared to clusters. Such a standby configuration would provide greater protection against user errors, system problems, and corruptions. Errors can appear on the primary database without propagating to the standby. In these cases, the only database copy not impacted would be on the standby database's disks.

If your environment demands fast failover as well as data protection, you can implement cluster solutions along with Automated Standby Database. Oracle Fail Safe or Oracle9*i* Real Application Clusters can be used for quick failover to another instance for transient faults, with failover to a standby reserved for more serious problems.

See Also: *Oracle9i Real Application Clusters Concepts* for more information about high availability strategy

Overview of Oracle9*i* Data Guard

Oracle9*i* provides a number of new features that help to prevent or minimize losses due to human errors, disasters, and data corruption.

Physical Standby Database

Physical standby database is the Oracle9*i* version of the Oracle8 Automated Standby Database feature, with one difference. The log transport services are now a separate component. The log transport services have been enhanced to support the new logical standby database feature and other features as well as physical standby database.

We call this "physical" standby because of this feature's roots in recovery. A physical standby is physically identical to the primary. Put another way, standby on-disk data structures are identical to the primary's on a block-for-block basis, because recovery applies changes block-for-block using the physical ROWID. The database schema, including indexes, must be the same, and the database cannot be opened read / write.

Logical Standby Database

Logical standby database is a new feature that takes standard Oracle archive logs, transforms them back into SQL transactions, and then applies them to an open standby database. Because the database is open, it is physically different from the primary database. As the standby is logically the same as the primary, it can be used to take over processing if the primary database is mistakenly harmed by human error, a corruption, or a disaster. Because transactions are applied using SQL, the standby database can be used concurrently for other tasks. Decision support can be optimized by using different indexes and materialized views than those on the primary.

Logical standby database is first and foremost a data protection feature. Just like physical standby database, it uses archive logs shipped the moment they are created on the primary system, performing all related processing on the standby database, out of harm's way in the case of a primary database failure. Updates recorded in the log always include the previous values as well as the new, updated values. Logical standby database compares these previous values to the previous values in the logical standby database.

Log Transport Services

Log transport services are used by both physical and logical standby database components. The functions it provides include control of different log shipping mechanisms, log shipping error handling and reporting, and retrieving "lost" logs after a system failure. Guaranteed data protection is now possible using one of the new log transport modes.

Oracle9*i* Data Guard Broker

Data Guard broker provides monitoring, control, and automation of the log transport services and the logical and physical standby components. For instance, using only one command to initiate failover, Data Guard broker can be used to control the entire process as the primary role moves from the primary to either type of standby database. Users can choose one of two different interfaces to perform role changes such as failover, that is, having the standby take over production processing from the primary database. One option is to use the new Oracle Enterprise Manager Data Guard Manager. It provides a graphical user interface (GUI) for most configuration and set-up tasks, as well as operational functions. A command-line interface (CLI) is also available. It provides access to both basic monitoring and all commands needed to make role changes, as well as the ability to configure and set up a Oracle9*i* Data Guard environment.

Oracle Enterprise Manager Data Guard Manager

Data Guard Manager is a part of Oracle Enterprise Manager. More than GUI-based access, the complete Oracle Enterprise Manager architecture is implemented while also fully integrated with Oracle9*i* Data Guard broker.

Oracle9*i* LogMiner™

LogMiner has been significantly enhanced in Oracle9*i*. LogMiner is a relational tool that lets the administrator use SQL to read, analyze, and interpret log files. LogMiner can view any redo log file, online or archived.

LogMiner technology now provides much of the infrastructure used by logical standby database and other features that are not discussed here, as well as broader data type support and other enhancements to LogMiner itself. A new Oracle Enterprise Manager application, Oracle9*i* LogMiner™ Viewer, adds a GUI-based interface to the pre-existing command-line interface.

See Also: "LogMiner SQL-Based Log Analyzer Overview" on page 2-43 for more information about LogMiner

How the Data Guard Components Work Together

Figure 2–5 illustrates how these components fit together, as described in the text following the figure.

Figure 2–5 Oracle9i Data Guard Component Architecture

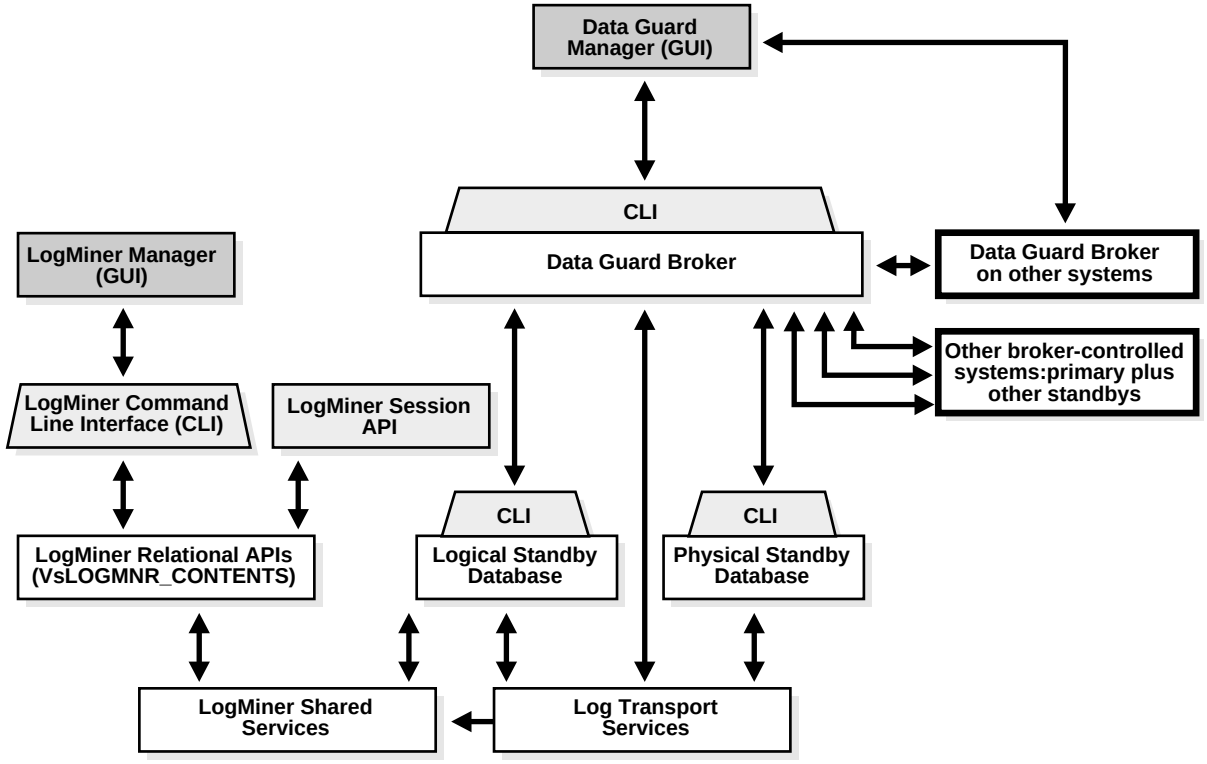


Figure 2–5 shows the various access options and user interface preferences.

- Logical standby database and physical standby database can be accessed as standalone components using command-line interfaces (CLIs) specific to each. Scripts written for prior versions of physical standby database can run because

existing commands and interfaces have been retained for compatibility. Alternatively, new DDL commands provide each with enhanced CLIs. Wherever possible, common syntax and semantics are used, minimizing the logical standby learning time required for a DBA familiar with physical standby.

- Data Guard broker controls the primary and all standby systems as well as their log transport. Accessing Data Guard using its command-line interface provides the following features:
  - Setup and configuration: The broker can be used to set up multiple systems once some prerequisite activities are performed on each of the individual systems.
  - Failover: By using the broker, operations like failover can be run using just a single command. The CLI can also be used to initiate failover. A unified view of status across all broker-monitored systems also can be queried using CLI.
  - Events and availability monitoring: Data Guard broker provides CLI-based access to a unified view of all systems' health and availability.
  - Access using any system: The user can connect to any system under the broker's control, enabling systems monitoring and control to continue operation despite system failures. Once connected, all broker-controlled systems are visible, and all operations can be performed. This eliminates a single point of failure for broker access.
- Oracle Enterprise Manager's Data Guard Manager also provides a multisystem view using the Data Guard broker. All of the broker's CLI-based functionality is provided by Data Guard Manager. Additionally, its graphical monitor provides visual cues such as color changes when critical events occur, more guidance for both set-up and configuration, and simpler execution of operations such as failover. Also, for a two-node physical standby configuration, an Oracle Enterprise Manager Data Guard Manager wizard leads users through the steps required to get up and running.

There are a number of factors to consider when choosing whether to implement a physical standby database, a logical standby database, or both. These include:

- Logical standby database can be used for dual purposes. While changes are being applied to the logical standby, its database can be open, servicing reporting requests at the same time.
- Logical standby database requires a higher level of investment in database administrator expertise.

- The solution that maximizes data protection against all corruptions typically includes both logical and physical standby databases.

  See Also: *Oracle9i Data Guard Concepts and Administration* for more information about Data Guard

Disaster Recovery Server and DRMON

In today's world of e-commerce, companies that do business on the Internet must have a strategy for recovering their applications and databases when, not if, things go wrong. The subject of disaster recovery and planned or unplanned failover should be considered by every database administrator who might oversee an unplanned outage from a real disaster such as a hurricane, or a planned outage such as when upgrading hardware.

The Disaster Recovery (DR) Server is part of a larger high availability strategy that helps the database administrator achieve this goal.

See Also:

- *Oracle9i Real Application Clusters Concepts* for more information about high availability strategy
- *Oracle9i Data Guard Concepts and Administration* for information about Data Guard

The DR Server is a distributed computing system that delivers enhanced ability to recover from various disaster scenarios.

An Oracle DR Server fundamentally consists of a collection of loosely connected nodes that combines physical and logical standby solutions into a single, easily managed disaster recovery solution. DR Server nodes, or sites, may be dispersed geographically and, if so, are connected through the network. Each DR Server node may be a simple instance, or the node may be a more complex system such as a fail safe cluster. DR Server manages these nodes as a single distributed computing system that offers higher availability than one can get from the individual nodes alone.

DR Server implements a failover system with replicated data across the server nodes. The database administrator configures the server such that the database and applications are instantiated on each node. One node is designated a primary node. This is where the database is fully available to applications and from whence data is shipped primarily in the form of redo logs. The other server nodes serve as standby

systems to the primary node. They accept redo logs shipped from the primary node and apply changes (logically or physically) to their copies of the database.

The standby nodes of the DR Server are ready to take over in the event that the primary node fails. Thus, the data and applications being served to users remain available in the face of a disaster that removes the primary node from operation.

The DR Server architecture provides two major functions for the database administrator:

- It provides a way for the DBA to logically configure a failover grouping of resources to be managed as a highly available service.
- It also specifies the underlying computing framework that constitutes the operational DR Server system itself.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for more information about the disaster recovery server and DRMON

Limiting Data Loss

In a primary / standby configuration, all noncurrent logs of the primary site are shipped to the standby site. This keeps the standby site up to date. However, if the primary database shuts down unexpectedly, records in a redo log that is still recording will be lost, because they have not yet been archived to the standby site. Open threads in a redo log file constitute a lag between the primary and standby databases.

Oracle9*i*, Release 1 (9.0.1), provides some ways to limit this lag and the data loss it can cause.

- You have the option of having the LGWR process shipping redo log data to the standby database at the same time it writes them to the local disk. This functionality is called standby zero data loss. This method essentially provides remote redo log mirroring. However, there is a large performance loss.
- You can limit the number of redo records that can be lost between the primary and standby sites. You do this by setting the system parameter `ARCHIVE_LAG_TARGET` in the initialization file to the number of seconds that can elapse between the time a redo log is started and the time it is archived to standby. A recommended value is 1800 seconds (30 minutes).

Note: This value is a target, not a guaranteed limit. After the primary site shuts down, any transaction whose commit record is missed by the standby is lost as a whole in the standby database. As a result, long transactions will make the standby data loss larger.

See Also: *Oracle9i Data Guard Concepts and Administration* for details about choosing values for `ARCHIVE_LAG_TARGET`

LogMiner SQL-Based Log Analyzer Overview

LogMiner is a relational tool that lets you read, analyze, and interpret online and archived log files using SQL. You can also use LogMiner Viewer to access LogMiner functionality. LogMiner Viewer, which is available with Oracle Enterprise Manager, provides a graphical user interface to LogMiner.

Analysis of the log files with LogMiner can be used to:

- Track specific sets of changes based on transaction, user, table, time, and so on. You can determine who modified a database object and what the object data was before and after the modification. The ability to trace and audit database changes back to their source and undo the changes provides data security and control.
- Pinpoint when an incorrect modification was introduced into the database. This lets you perform logical recovery at the application level instead of at the database level.
- Provide supplemental information for tuning and capacity planning. You can also perform historical analysis to determine trends and data access patterns.
- Retrieve critical information for debugging complex applications.

Note: LogMiner can only read and analyze log files from Oracle8 or later.

See Also: *Oracle9i Database Administrator's Guide* for more information about the LogMiner

Part II

Database Structures

Part II describes the basic structural architecture of the Oracle database, including physical and logical storage structures. Part II contains the following chapters:

- Chapter 3, "Data Blocks, Extents, and Segments"
- Chapter 4, "Tablespaces, Datafiles, and Control Files"
- Chapter 5, "The Data Dictionary"

3

Data Blocks, Extents, and Segments

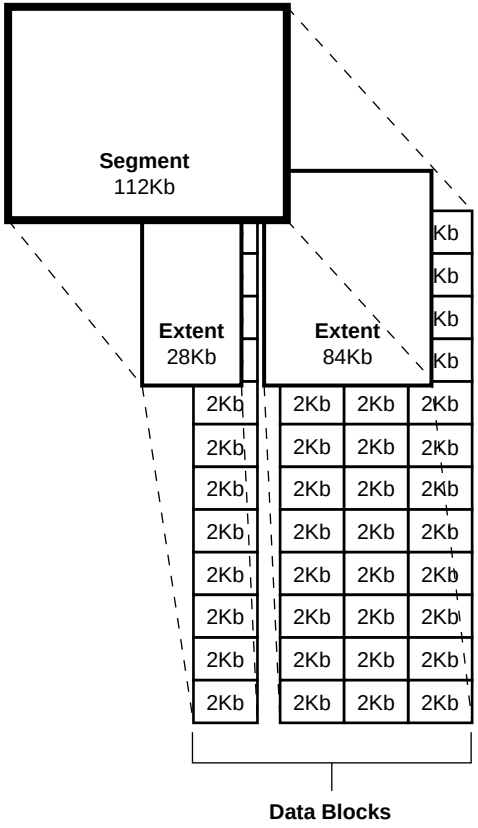
This chapter describes the nature of and relationships among the logical storage structures in the Oracle server. It includes:

- Introduction to Data Blocks, Extents, and Segments
- Data Blocks Overview
- Extents Overview
- Segments Overview
- Automatic Undo Management

Introduction to Data Blocks, Extents, and Segments

Oracle allocates logical database space for all data in a database. The units of database space allocation are data blocks, extents, and segments. Figure 3–1 shows the relationships among these data structures:

Figure 3–1 The Relationships Among Segments, Extents, and Data Blocks



At the finest level of granularity, Oracle stores data in **data blocks** (also called **logical blocks**, **Oracle blocks**, or **pages**). One data block corresponds to a specific number of bytes of physical database space on disk.

The next level of logical database space is an **extent**. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.

The level of logical database storage above an extent is called a **segment**. A segment is a set of extents, each of which has been allocated for a specific data structure and all of which are stored in the same tablespace. For example, each table's data is stored in its own **data segment**, while each index's data is stored in its own **index segment**. If the table or index is partitioned, each partition is stored in its own segment.

Oracle allocates space for segments in units of one extent. When the existing extents of a segment are full, Oracle allocates another extent for that segment. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on disk.

A segment and all its extents are stored in one tablespace. Within a tablespace, a segment can include extents from more than one file; that is, the segment can span datafiles. However, each extent can contain data from only one datafile.

Although you can allocate additional extents, the blocks themselves are allocated separately. If you allocate an extent to a specific instance, the blocks are immediately allocated to the free list. However, if the extent is not allocated to a specific instance, then the blocks themselves are allocated only when the high water mark moves. The **high water mark** is the boundary between used and unused space in a segment.

Note: In Oracle9*i*, Release 1 (9.0.1), free space can be managed automatically. See "Free Space Management" on page 3-6.

Data Blocks Overview

Oracle manages the storage space in the datafiles of a database in units called **data blocks**. A data block is the smallest unit of I/O used by a database. In contrast, at the physical, operating system level, all data is stored in bytes. Each operating system has a **block size**. Oracle requests data in multiples of Oracle data blocks, not operating system blocks.

The standard block size is specified by the initialization parameter `DB_BLOCK_SIZE`. In addition, Oracle9*i*, Release 1 (9.0.1), permits specification of up to five nonstandard block sizes. The data block sizes should be a multiple of the operating system's block size within the maximum limit to avoid unnecessary I/O. Oracle data blocks are the smallest units of storage that Oracle can use or allocate.

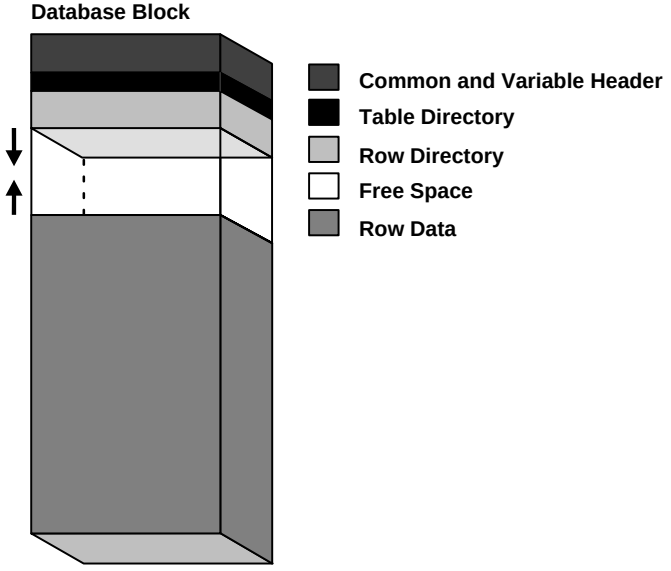
See Also:

- Your Oracle operating system specific documentation for more information about data block sizes
- "Nonstandard Block Sizes" on page 4-13 for information about using nonstandard block sizes

Data Block Format

The Oracle data block format is similar regardless of whether the data block contains table, index, or clustered data. Figure 3–2 illustrates the format of a data block.

Figure 3–2 Data Block Format



Header (Common and Variable)

The header contains general block information, such as the block address and the type of segment (for example, data, index, or rollback).

Table Directory

This portion of the data block contains information about the table having rows in this block.

Row Directory

This portion of the data block contains information about the actual rows in the block (including addresses for each row piece in the row data area).

Once the space has been allocated in the row directory of a data block's overhead, this space is not reclaimed when the row is deleted. Therefore, a block that is currently empty but had up to 50 rows at one time continues to' have 100 bytes allocated in the header for the row directory. Oracle reuses this space only when new rows are inserted in the block.

Overhead

The data block header, table directory, and row directory are referred to collectively as **overhead**. Some block overhead is fixed in size; the total block overhead size is variable. On average, the fixed and variable portions of data block overhead total 84 to 107 bytes.

Row Data

This portion of the data block contains table or index data. Rows can span blocks.

See Also: "Row Chaining and Migrating" on page 3-7

Free Space

Free space is allocated for insertion of new rows and for updates to rows that require additional space (for example, when a trailing null is updated to a nonnull value). Whether issued insertions actually occur in a given data block is a function of current free space in that data block and the value of the space management parameter `PCTFREE`.

In data blocks allocated for the data segment of a table or cluster, or for the index segment of an index, free space can also hold transaction entries. A **transaction entry** is required in a block for each `INSERT`, `UPDATE`, `DELETE`, and `SELECT...FOR UPDATE` statement accessing one or more rows in the block. The space required for transaction entries is operating system dependent; however, transaction entries in most operating systems require approximately 23 bytes.

See Also:

- "Free Space Management" on page 3-6 for an introduction to in-segment free space management
- "PCTFREE, PCTUSED, and Row Chaining" on page 3-8 for more information about space management using free lists

Free Space Management

Free space can be managed automatically or manually.

With Oracle9*i*, Release 1 (9.0.1), free space can be managed automatically inside database segments. The in-segment free/used space is tracked using bitmaps, as opposed to free lists. Use of automatic segment-space management offers the following benefits:

- Ease of use
- Better space utilization, especially for the objects with highly varying size rows
- Better run-time adjustment to variations in concurrent access
- Better multi-instance behavior in terms of performance/space utilization
- Preparation for future enhancements, such as in-space segment reorganization and in-place tablespace reorganization

You specify automatic segment-space management when you create a locally managed tablespace. The specification then applies to all segments subsequently created in this tablespace.

See Also:

- *Oracle9i Database Administrator's Guide*
- *Oracle9i SQL Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

Availability and Compression of Free Space in a Data Block

Two types of statements can increase the free space of one or more data blocks: `DELETE` statements, and `UPDATE` statements that update existing values to smaller values. The released space from these types of statements is available for subsequent `INSERT` statements under the following conditions:

- If the INSERT statement is in the same transaction and subsequent to the statement that frees space, the INSERT statement can use the space made available.
- If the INSERT statement is in a separate transaction from the statement that frees space (perhaps being executed by another user), the INSERT statement can use the space made available only after the other transaction commits and only if the space is needed.

Released space may or may not be contiguous with the main area of free space in a data block. Oracle coalesces the free space of a data block *only* when (1) an INSERT or UPDATE statement attempts to use a block that contains enough free space to contain a new row piece, and (2) the free space is fragmented so the row piece cannot be inserted in a contiguous section of the block. Oracle does this compression only in such situations, because otherwise the performance of a database system decreases due to the continuous compression of the free space in data blocks.

Row Chaining and Migrating

In two circumstances, the data for a row in a table may be too large to fit into a single data block. In the first case, the row is too large to fit into one data block when it is first inserted. In this case, Oracle stores the data for the row in a **chain** of data blocks (one or more) reserved for that segment. Row chaining most often occurs with large rows, such as rows that contain a column of datatype LONG or LONG RAW. Row chaining in these cases is unavoidable.

However, in the second case, a row that originally fit into one data block is updated so that the overall row length increases, and the block's free space is already completely filled. In this case, Oracle **migrates** the data for the entire row to a new data block, assuming the entire row can fit in a new block. Oracle preserves the original row piece of a migrated row to point to the new block containing the migrated row. The rowid of a migrated row does not change.

When a row is chained or migrated, I/O performance associated with this row decreases because Oracle must scan more than one data block to retrieve the information for the row.

See Also:

- "Row Format and Size" on page 11-6 for more information on the format of a row and a row piece
- "Rowids of Row Pieces" on page 11-8 for more information on rowids
- "Physical Rowids" on page 13-18 for information about rowids
- *Oracle9i Database Performance Guide and Reference* for information about reducing chained and migrated rows and improving I/O performance

PCTFREE, PCTUSED, and Row Chaining

For manually managed tablespaces, two space management parameters, PCTFREE and PCTUSED, enable you to control the use of free space for inserts of and updates to the rows in all the data blocks of a particular segment. Specify these parameters when you create or alter a table or cluster (which has its own data segment). You can also specify the storage parameter PCTFREE when creating or altering an index (which has its own index segment).

Note: This discussion does not apply to LOB datatypes (BLOB, CLOB, NCLOB, and BFILE). They do not use the PCTFREE storage parameter or free lists.

See "LOB Datatypes" on page 13-14 for more information.

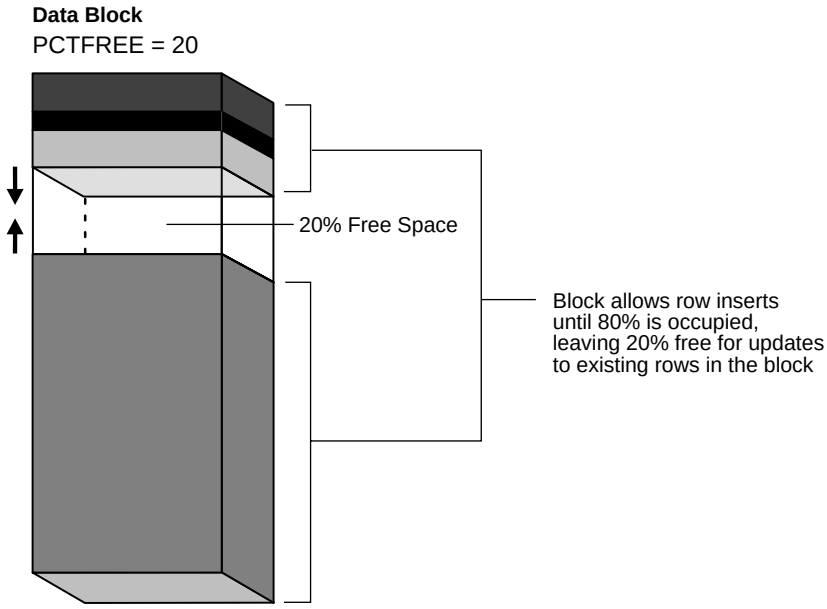
The PCTFREE Parameter

The PCTFREE parameter sets the minimum percentage of a data block to be **reserved** as free space for possible updates to rows that already exist in that block. For example, assume that you specify the following parameter within a CREATE TABLE statement:

```
PCTFREE 20
```

This states that 20% of each data block in this table's data segment will be kept free and available for possible updates to the existing rows already within each block. New rows can be added to the row data area, and corresponding information can be added to the variable portions of the overhead area, until the row data and overhead total 80% of the total block size. Figure 3–3 illustrates PCTFREE.

Figure 3–3 PCTFREE



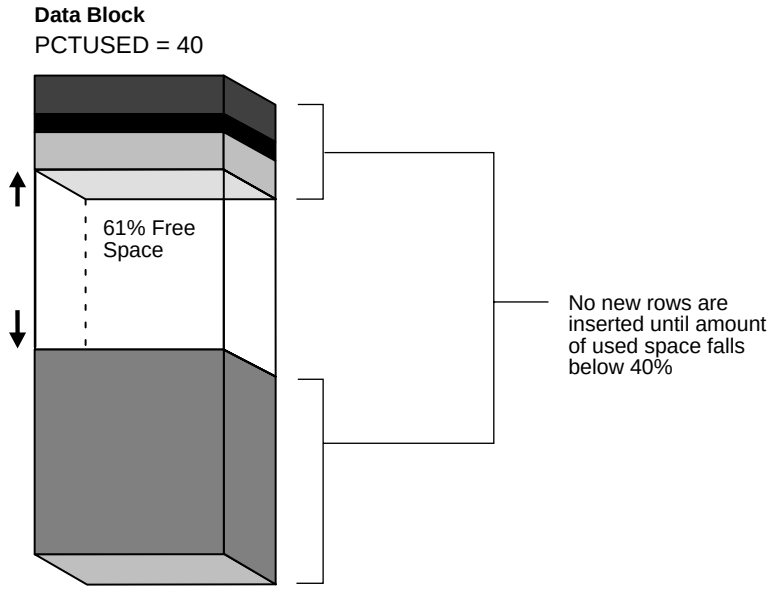
The PCTUSED Parameter

The PCTUSED parameter sets the minimum percentage of a block that can be used for row data plus overhead before new rows are added to the block. After a data block is filled to the limit determined by PCTFREE, Oracle considers the block unavailable for the insertion of new rows until the percentage of that block falls below the parameter PCTUSED. Until this value is achieved, Oracle uses the free space of the data block only for updates to rows already contained in the data block. For example, assume that you specify the following parameter in a CREATE TABLE statement:

```
PCTUSED 40
```

In this case, a data block used for this table's data segment is considered unavailable for the insertion of any new rows until the amount of used space in the block falls to 39% or less (assuming that the block's used space has previously reached PCTFREE). Figure 3–4 illustrates this.

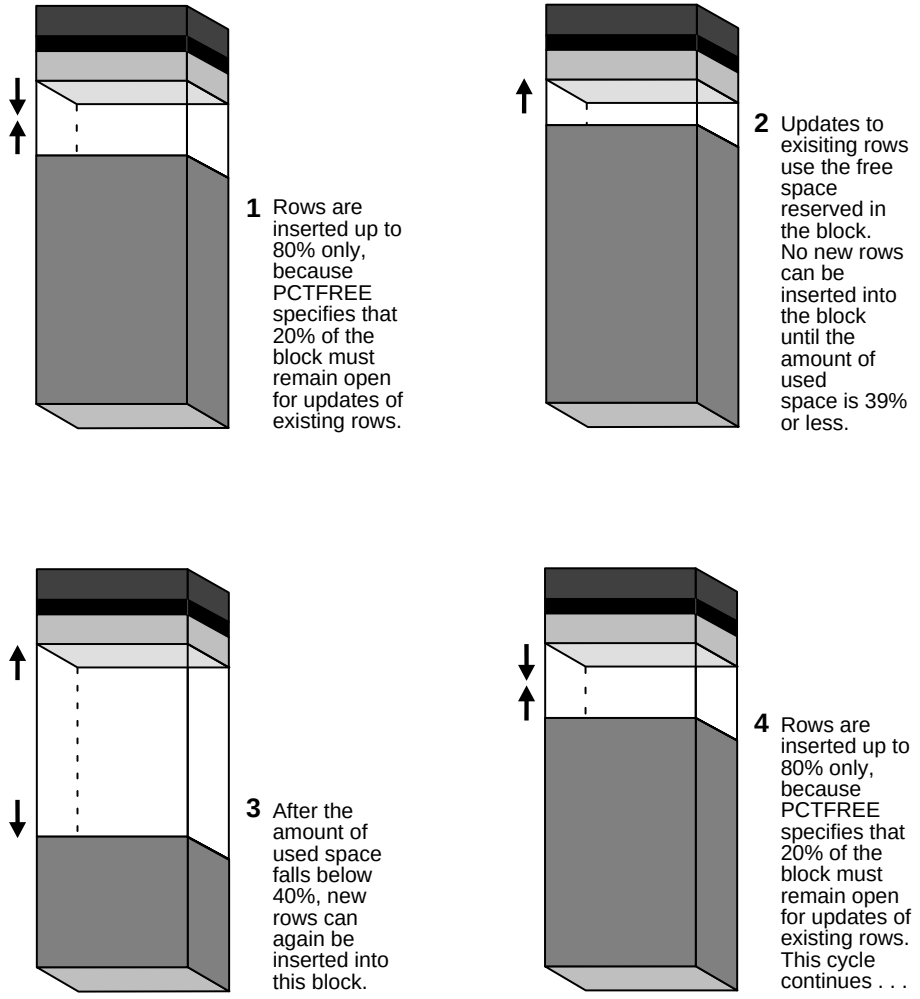
Figure 3–4 PCTUSED



How PCTFREE and PCTUSED Work Together

PCTFREE and PCTUSED work together to optimize the use of space in the data blocks of the extents within a data segment. Figure 3–5 illustrates the interaction of these two parameters.

Figure 3–5 Maintaining the Free Space of Data Blocks with PCTFREE and PCTUSED



In a newly allocated data block, the space available for inserts is the block size minus the sum of the block overhead and free space (PCTFREE). Updates to existing data can use any available space in the block. Therefore, updates can reduce the available space of a block to less than PCTFREE, the space reserved for updates but not accessible to inserts.

For each data and index segment, Oracle maintains one or more **free lists**—lists of data blocks that have been allocated for that segment's extents and have free space greater than `PCTFREE`. These blocks are available for inserts. When you issue an `INSERT` statement, Oracle checks a free list of the table for the first available data block and uses it if possible. If the free space in that block is not large enough to accommodate the `INSERT` statement, and the block is at least `PCTUSED`, Oracle takes the block off the free list. Multiple free lists for each segment can reduce contention for free lists when concurrent inserts take place.

After you issue a `DELETE` or `UPDATE` statement, Oracle processes the statement and checks to see if the space being used in the block is now less than `PCTUSED`. If it is, the block goes to the beginning of the transaction free list, and it is the first of the available blocks to be used in that transaction. When the transaction commits, free space in the block becomes available for other transactions.

Extents Overview

An extent is a logical unit of database storage space allocation made up of a number of contiguous data blocks. One or more extents in turn make up a segment. When the existing space in a segment is completely used, Oracle allocates a new extent for the segment.

When Extents Are Allocated

When you create a table, Oracle allocates to the table's data segment an **initial extent** of a specified number of data blocks. Although no rows have been inserted yet, the Oracle data blocks that correspond to the initial extent are reserved for that table's rows.

If the data blocks of a segment's initial extent become full and more space is required to hold new data, Oracle automatically allocates an **incremental extent** for that segment. An incremental extent is a subsequent extent of the same or greater size than the previously allocated extent in that segment.

For maintenance purposes, the header block of each segment contains a directory of the extents in that segment.

Rollback segments always have at least two extents.

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

See Also: "How Extents Are Deallocated from a Rollback Segment" on page 3-30

Determine the Number and Size of Extents

Storage parameters expressed in terms of extents define every segment. Storage parameters apply to all types of segments. They control how Oracle allocates free database space for a given segment. For example, you can determine how much space is initially reserved for a table's data segment or you can limit the number of extents the table can allocate by specifying the storage parameters of a table in the `STORAGE` clause of the `CREATE TABLE` statement. If you do not specify a table's storage parameters, it uses the default storage parameters of the tablespace.

Tablespaces can manage their extents either locally or through the data dictionary. Some storage parameters apply only to extents in dictionary-managed tablespaces, and other storage parameters apply to all extents.

See Also: "Space Management in Tablespaces" on page 4-11

Extents Managed Locally

A tablespace that manages its extents locally can have either uniform extent sizes or variable extent sizes that are determined automatically by the system. When you create the tablespace, the `UNIFORM` or `AUTOALLOCATE` (system-managed) clause specifies the type of allocation.

- For system-managed extents, you can specify the size of the initial extent and Oracle determines the optimal size of additional extents, with a minimum extent size of 64 KB. This is the default for permanent tablespaces.
- For uniform extents, you can specify an extent size or use the default size, which is 1 MB. Temporary tablespaces that manage their extents locally can only use this type of allocation.

The storage parameters `NEXT, PCTINCREASE, MINEXTENTS, MAXEXTENTS,` and `DEFAULT STORAGE` are not valid for extents that are managed locally.

Extents Managed by the Data Dictionary

A tablespace that uses the data dictionary to manage its extents has incremental extent sizes, which are determined by the storage parameters `INITIAL, NEXT,` and `PCTINCREASE.` When you create a schema object in the tablespace, its first extent is allocated with the `INITIAL` size. When additional space is needed, the `NEXT` and `PCTINCREASE` parameters determine the sizes of new extents. You can modify the values of `NEXT` and `PCTINCREASE` after creating a schema object.

See Also:

- *Oracle9i Database Administrator's Guide*
- *Oracle9i SQL Reference*

for more information about storage parameters

How Extents Are Allocated

Oracle uses different algorithms to allocate extents, depending on whether they are locally managed or dictionary managed.

Allocating Extents in Locally Managed Tablespaces

In locally managed tablespaces, Oracle looks for free space to allocate to a new extent by first determining a candidate datafile in the tablespace and then searching the datafile's bitmap for the required number of adjacent free blocks. If that datafile does not have enough adjacent free space, Oracle looks in another datafile.

Allocating Extents in Dictionary-Managed Tablespaces

In dictionary-managed tablespaces, Oracle controls the allocation of incremental extents for a given segment as follows:

1. Oracle searches through the free space (in the tablespace that contains the segment) for the first free, contiguous set of data blocks of an incremental extent's size or larger, using the following algorithm:

   a. Oracle searches for a contiguous set of data blocks that matches the size of new extent plus one block to reduce internal fragmentation. (The size is rounded up to the size of the minimal extent for that tablespace, if necessary.) For example, if a new extent requires 19 data blocks, Oracle searches for exactly 20 contiguous data blocks. If the new extent is 5 or fewer blocks, Oracle does not add an extra block to the request.

   b. If an exact match is not found, Oracle then searches for a set of contiguous data blocks greater than the amount needed. If Oracle finds a group of contiguous blocks that is at least 5 blocks greater than the size of the extent needed, it splits the group of blocks into separate extents, one of which is the size it needs. If Oracle finds a group of blocks that is larger than the size it needs, but less than 5 blocks larger, it allocates all the contiguous blocks to the new extent.

      In the current example, if Oracle does not find a set of exactly 20 contiguous data blocks, Oracle searches for a set of contiguous data blocks greater than 20. If the first set it finds contains 25 or more blocks, it breaks the blocks up and allocates 20 of them to the new extent and leaves the remaining 5 or more blocks as free space. Otherwise, it allocates all of the blocks (between 21 and 24) to the new extent.

   c. If Oracle does not find an equal or larger set of contiguous data blocks, it coalesces any free, adjacent data blocks in the corresponding tablespace to form larger sets of contiguous data blocks. (The SMON background process also periodically coalesces adjacent free space.) After coalescing a tablespace's data blocks, Oracle performs the searches described in 1a and 1b again.

   d. If an extent cannot be allocated after the second search, Oracle tries to resize the files by autoextension. If Oracle cannot resize the files, it returns an error.

2. Once Oracle finds and allocates the necessary free space in the tablespace, it allocates a portion of the free space that corresponds to the size of the incremental extent. If Oracle found a larger amount of free space than was required for the extent, Oracle leaves the remainder as free space (no smaller than 5 contiguous blocks).
3. Oracle updates the segment header and data dictionary to show that a new extent has been allocated and that the allocated space is no longer free.

The blocks of a newly allocated extent, although they were free, may not be empty of old data. Usually, Oracle formats the blocks of a newly allocated extent when it starts using the extent, but only as needed (starting with the blocks on the segment free list). In a few cases, however, such as when a database administrator forces allocation of an incremental extent with the `ALLOCATE EXTENT` clause of an `ALTER TABLE` or `ALTER CLUSTER` statement, Oracle formats the extent's blocks when it allocates the extent.

When Extents Are Deallocated

In general, the extents of a segment do not return to the tablespace until you drop the schema object whose data is stored in the segment (using a `DROP TABLE` or `DROP CLUSTER` statement). Exceptions to this include the following:

- The owner of a table or cluster, or a user with the `DELETE ANY` privilege, can truncate the table or cluster with a `TRUNCATE...DROP STORAGE` statement.
- Periodically, Oracle deallocates one or more extents of a rollback segment if it has the `OPTIMAL` size specified.
- A database administrator (DBA) can deallocate unused extents using the following SQL syntax:

  ```
  ALTER TABLE table_name DEALLOCATE UNUSED;
  ```

When extents are freed, Oracle modifies the bitmap in the datafile (for locally managed tablespaces) or updates the data dictionary (for dictionary-managed tablespaces) to reflect the regained extents as available space. Any data in the blocks of freed extents becomes inaccessible, and Oracle clears the data when the blocks are subsequently reused for other extents.

See Also:

- *Oracle9i Database Administrator's Guide*
- *Oracle9i SQL Reference*

for more information about deallocating extents

Extents in Nonclustered Tables

As long as a nonclustered table exists or until you truncate the table, any data block allocated to its data segment remains allocated for the table. Oracle inserts new rows into a block if there is enough room. Even if you delete all rows of a table, Oracle does not reclaim the data blocks for use by other objects in the tablespace.

After you drop a nonclustered table, this space can be reclaimed when other extents require free space. Oracle reclaims all the extents of the table's data and index segments for the tablespaces that they were in and makes the extents available for other schema objects in the same tablespace.

In dictionary-managed tablespaces, when a segment requires an extent larger than the available extents, Oracle identifies and combines contiguous reclaimed extents to form a larger one. This is called **coalescing** extents.

Coalescing extents is not necessary in locally managed tablespaces, because all contiguous free space is available for allocation to a new extent regardless of whether it was reclaimed from one or more extents.

Extents in Clustered Tables

Clustered tables store their information in the data segment created for the cluster. Therefore, if you drop one table in a cluster, the data segment remains for the other tables in the cluster, and no extents are deallocated. You can also truncate clusters (except for hash clusters) to free extents.

Extents in Materialized Views and Their Logs

Oracle deallocates the extents of materialized views and materialized view logs in the same manner as for tables and clusters.

See Also: "Materialized Views" on page 11-21 for a description of materialized views and their logs

Extents in Indexes

All extents allocated to an index segment remain allocated as long as the index exists. When you drop the index or associated table or cluster, Oracle reclaims the extents for other uses within the tablespace.

Extents in Rollback Segments

Oracle periodically checks the rollback segments of the database to see if they have grown larger than their optimal size. If a rollback segment is larger than is optimal (that is, it has too many extents), Oracle automatically deallocates one or more extents from the rollback segment.

> **See Also:** "How Extents Are Deallocated from a Rollback Segment" on page 3-30

Extents in Temporary Segments

When Oracle completes the execution of a statement requiring a temporary segment, Oracle automatically drops the temporary segment and returns the extents allocated for that segment to the associated tablespace. A single sort allocates its own temporary segment in the temporary tablespace of the user issuing the statement and then returns the extents to the tablespace.

Multiple sorts, however, can use sort segments in a temporary tablespace designated exclusively for sorts. These sort segments are allocated only once for the instance, and they are not returned after the sort but remain available for other multiple sorts.

A temporary segment in a temporary table contains data for multiple statements of a single transaction or session. Oracle drops the temporary segment at the end of the transaction or session, returning the extents allocated for that segment to the associated tablespace.

> **See Also:**
>
> - "Introduction to Temporary Segments" on page 3-20
> - "Temporary Tables" on page 11-12

Segments Overview

A segment is a set of extents that contains all the data for a specific logical storage structure within a tablespace. For example, for each table, Oracle allocates one or

more extents to form that table's data segment, and for each index, Oracle allocates one or more extents to form its index segment.

Oracle databases use four types of segments, which are described in the following sections:

- Introduction to Data Segments
- Introduction to Index Segments
- Introduction to Temporary Segments
- Introduction to Rollback Segments

Introduction to Data Segments

A single data segment in an Oracle database holds all of the data for one of the following:

- A table that is not partitioned or clustered
- A partition of a partitioned table
- A cluster of tables

Oracle creates this data segment when you create the table or cluster with the `CREATE` statement.

The storage parameters for a table or cluster determine how its data segment's extents are allocated. You can set these storage parameters directly with the appropriate `CREATE` or `ALTER` statement. These storage parameters affect the efficiency of data retrieval and storage for the data segment associated with the object.

Note: Oracle creates segments for materialized views and materialized view logs in the same manner as for tables and clusters.

See Also:

- *Oracle9i Replication* for information on materialized views and materialized view logs
- *Oracle9i SQL Reference* for information on the `CREATE` and `ALTER` statements

Introduction to Index Segments

Every nonpartitioned index in an Oracle database has a single index segment to hold all of its data. For a partitioned index, every partition has a single index segment to hold its data.

Oracle creates the index segment for an index or an index partition when you issue the `CREATE INDEX` statement. In this statement, you can specify storage parameters for the extents of the index segment and a tablespace in which to create the index segment. (The segments of a table and an index associated with it do not have to occupy the same tablespace.) Setting the storage parameters directly affects the efficiency of data retrieval and storage.

Introduction to Temporary Segments

When processing queries, Oracle often requires temporary workspace for intermediate stages of SQL statement parsing and execution. Oracle automatically allocates this disk space called a **temporary segment**. Typically, Oracle requires a temporary segment as a work area for sorting. Oracle does not create a segment if the sorting operation can be done in memory or if Oracle finds some other way to perform the operation using indexes.

Operations that Require Temporary Segments

The following statements could require the use of a temporary segment:

- `CREATE INDEX`
- `SELECT ... ORDER BY`
- `SELECT DISTINCT ...`
- `SELECT ... GROUP BY`
- `SELECT ... UNION`
- `SELECT ... INTERSECT`
- `SELECT ... MINUS`

Some unindexed joins and correlated subqueries can also require use of a temporary segment. For example, if a query contains a `DISTINCT` clause, a `GROUP BY`, and an `ORDER BY`, Oracle can require as many as two temporary segments. If applications often issue statements in the previous list, the database administrator can improve performance by adjusting the initialization parameter `SORT_AREA_SIZE`.

See Also: *Oracle9i Database Reference* for information on SORT_AREA_SIZE and other initialization parameters

Segments in Temporary Tables and Their Indexes

Oracle can also allocate temporary segments for temporary tables and indexes created on temporary tables. Temporary tables hold data that exists only for the duration of a transaction or session.

See Also: "Temporary Tables" on page 11-12

How Temporary Segments Are Allocated

Oracle allocates temporary segments differently for queries and temporary tables.

Allocation of Temporary Segments for Queries Oracle allocates temporary segments as needed during a user session, in the temporary tablespace of the user issuing the statement. Specify this tablespace with a CREATE USER or an ALTER USER statement using the TEMPORARY TABLESPACE clause. If no temporary tablespace has been defined for the user, the default temporary tablespace is the SYSTEM tablespace. The default storage characteristics of the containing tablespace determine those of the extents of the temporary segment.

Oracle drops temporary segments when the statement completes.

Because allocation and deallocation of temporary segments occur frequently, it is reasonable to create a special tablespace for temporary segments. By doing so, you can distribute I/O across disk devices, and you can avoid fragmentation of the SYSTEM and other tablespaces that otherwise hold temporary segments.

Entries for changes to temporary segments used for sort operations are not stored in the redo log, except for space management operations on the temporary segment.

See Also: Chapter 24, "Controlling Database Access" for more information about assigning a user's temporary segment tablespace

Allocation of Temporary Segments for Temporary Tables and Indexes Oracle allocates segments for a temporary table when the first INSERT into that table is issued. (This can be an insert operation internally issued by 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 the temporary tablespace of the user who created the temporary table.

Oracle drops segments for a transaction-specific temporary table at the end of the transaction and drops segments for a session-specific temporary table at the end of the session. If other transactions or sessions share the use of that temporary table, the segments containing their data remain in the table.

> **See Also:** "Temporary Tables" on page 11-12

Automatic Undo Management

In Oracle8*i*, undo space management was performed using rollback segments, as described in "Introduction to Rollback Segments" on page 3-24. This method is now called manual undo management mode.

Oracle9*i*, Release 1 (9.0.1), provides automatic undo management, a new way of managing undo space in Oracle databases. You need to perform fewer explicit actions to configure the system to use undo space efficiently and effectively. Automatic undo management is undo-tablespace based. You allocate space in the form of a few undo tablespaces, instead of allocating many rollback segments in different sizes. With automatic undo management, you have no reason to create, drop, or alter transaction tables or undo segments. You just need to create an undo tablespace. Undo management becomes much simpler.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about creating an undo tablespace

Automatic undo management gives you a way to explicitly control undo retention. Through the use of a system parameter (`UNDO_RETENTION`), you can specify the amount of committed undo information to retain in the database. You specify the parameter as clock time (for example, 30 seconds). With retention control, you can configure your system to enable long queries to run successfully.

You can use the performance view `V$UNDOSTAT` to monitor and configure your database system to achieve efficient use of undo space. `V$UNDOSTAT` shows various undo/transaction statistics, such as the amount of undo space consumed in the instance.

Undo Mode

Undo mode provides a more flexible way to migrate from manual undo management to automatic undo management. Starting with Oracle9*i*, Release 1 (9.0.1), a database system can run in either manual undo management mode or

automatic undo management mode. In manual undo management mode, undo space is managed through rollback segments, the way it was in Oracle8*i*. Manual undo management mode is supported under any compatibility level. Use it when you need to run Oracle9*i* to take advantage of some new features, but are not yet not ready to convert to automatic undo management mode.

In automatic undo management mode, undo space is managed in undo tablespaces. To use automatic undo management mode, the DBA needs only to create an undo tablespace for each instance and set the `UNDO_MANAGEMENT` initialization parameter to `AUTO`. Automatic undo management mode is supported under compatibility levels of Oracle9*i*, Release 1 (9.0.1) or higher. Although manual undo management mode is supported, you are strongly encouraged to run in automatic undo management mode.

> **See Also:** *Oracle9i Database Administrator's Guide* for descriptions of the syntax and the semantics of the DDL statements.

Undo Quota

In automatic undo management mode, the system controls exclusively the assignment of transactions to undo segments, and controls space allocation for undo segments. An ill-behaved transaction can potentially use up much of the undo space, thus paralyzing the entire system. In manual undo management mode, you can control such possibilities by limiting the size of rollback segments with small `MAXEXTENTS` values. However, you then have to explicitly assign long running transactions to larger rollback segments, using the `SET TRANSACTION USE ROLLBACK SEGMENT` statement. This approach has proven to be cumbersome.

A new Resource Manager directive, `UNDO_POOL`, is a more explicit way to control runaway transactions. The `UNDO_POOL` directive enables DBAs to group users into consumer groups, with each group assigned a maximum undo space limit. Whenever the total undo space consumed by a group exceeds the limit, its users are not allowed to make any further updates, until undo space is freed up by other members (after their transactions commit or abort).

The default value of `UNDO_POOL` is `UNLIMITED`, where users are allowed to consume as much undo space as the undo tablespace has. DBAs have the option of limiting a particular user by using the `UNDO_POOL` directive.

Undo Retention Control

Long-running queries sometimes fail because undo information required for consistent read operations is no longer available. This happens because committed undo blocks are overwritten by active transactions.

Automatic undo management provides a way to explicitly control when undo space can be reused—how long undo information will be retained. A DBA can specify a retention period by using the parameter, UNDO_RETENTION. For example, if UNDO_RETENTION is set to 30 minutes, all committed undo information in the system will be retained for at least 30 minutes. This ensures that all queries running for thirty minutes or less will not get the OER (snapshot too old) error, under normal circumstances.

You can either set UNDO_RETENTION at startup or change it dynamically with the ALTER SYSTEM statement. The following example sets retention to 20 minutes:

```
ALTER SYSTEM SET UNDO_RETENTION = 1200;
```

If you do not set the UNDO_RETENTION parameter, Oracle uses a small default value that should be adequate for most OLTP systems, where queries are not usually not very long.

In general, it is a good idea not to set retention to a value very close to what the undo tablespace can support, because that may result in excessive movement of space between undo segments. A 20% buffer of undo space is recommended.

External Views

You can monitor transaction and undo usage information with V$TRANSACTION and V$ROLLSTAT. For automatic undo management, the information in V$ROLLSTAT reflects the behaviors of the automatic undo management undo segments.

Oracle9*i*, Release 1 (9.0.1), adds a new view, V$UNDOSTAT. This view displays a histogram of statistical data to show how well the system is working. You can see statistics such as undo consumption rate, transaction concurrency, and lengths of queries executed in the instance. Using this view, you can better estimate the amount of undo space required for the current workload. This view is available in both the automatic undo management and manual undo management mode.

> **See Also:** *Oracle9i Database Administrator's Guide* for more details about using V$UNDOSTAT.

Introduction to Rollback Segments

Each database contains one or more rollback segments. A rollback segment records the old values of data that were changed by each transaction (whether or not committed). Rollback segments are used to provide read consistency, to roll back transactions, and to recover the database.

Note: You are strongly urged to use Oracle9*i*'s automatic undo management. This section is included only for backward compatibility with previous releases.

See Also:

- "Automatic Undo Management" on page 3-22
- "Multiversion Concurrency Control" on page 22-4 for information about read consistency
- "Rollback of Transactions" on page 17-7

Contents of a Rollback Segment

Information in a rollback segment consists of several **rollback entries**. Among other information, a rollback entry includes block information (the file number and block ID corresponding to the data that was changed) and the data as it existed before an operation in a transaction. Oracle links rollback entries for the same transaction, so the entries can be found easily if necessary for transaction rollback.

Neither database users nor administrators can access or read rollback segments. Only Oracle can write to or read them. (They are owned by the user SYS, no matter which user creates them.)

How Rollback Entries Are Logged

Rollback entries change data blocks in the rollback segment, and Oracle records all changes to data blocks, including rollback entries, in the redo log. This second recording of the rollback information is very important for active transactions (not yet committed or rolled back) at the time of a system crash. If a system crash occurs, Oracle automatically restores the rollback segment information, including the rollback entries for active transactions, as part of instance or media recovery. Once the recovery is complete, Oracle performs the actual rollbacks of transactions that had been neither committed nor rolled back at the time of the system crash.

When Rollback Information Is Required

For each rollback segment, Oracle maintains a **transaction table**—a list of all transactions that use the associated rollback segment and the rollback entries for each change performed by these transactions. Oracle uses the rollback entries in a rollback segment to perform a transaction rollback and to create read-consistent results for queries.

Rollback segments record the data prior to change for each transaction. For every transaction, Oracle links each new change to the previous change. If you must roll back the transaction, Oracle applies the changes in a chain to the data blocks in an order that restores the data to its previous state.

Similarly, when Oracle needs to provide a read-consistent set of results for a query, it can use information in rollback segments to create a set of data consistent with respect to a single point in time.

Transactions and Rollback Segments

Each time a user's transaction begins, the transaction is assigned to a rollback segment in one of two ways:

- Oracle can assign a transaction automatically to the next available rollback segment. The transaction assignment occurs when you issue the first DML or DDL statement in the transaction. Oracle never assigns read-only transactions (transactions that contain only queries) to a rollback segment, regardless of whether the transaction begins with a `SET TRANSACTION READ ONLY` statement.
- An application can assign a transaction explicitly to a specific rollback segment. At the start of a transaction, an application developer or user can specify a particular rollback segment that Oracle should use when executing the transaction. This lets the application developer or user select a large or small rollback segment, as appropriate for the transaction.

For the duration of a transaction, the associated user process writes rollback information only to the assigned rollback segment.

When you commit a transaction, Oracle releases the rollback information but does not immediately destroy it. The information remains in the rollback segment to create read-consistent views of pertinent data for queries that started before the transaction committed. To guarantee that rollback data is available for as long as possible for such views, Oracle writes the extents of rollback segments sequentially. When the last extent of the rollback segment becomes full, Oracle continues writing rollback data by wrapping around to the first extent in the segment. A long-running transaction (idle or active) can require a new extent to be allocated for the rollback segment.

See Figure 3–6 on page 3-28, Figure 3–7 on page 3-29, and Figure 3–8 on page 3-30 for more information about how transactions use the extents of a rollback segment.

Each rollback segment can handle a fixed number of transactions from one instance. Unless you explicitly assign transactions to particular rollback segments, Oracle

distributes active transactions across available rollback segments so that all rollback segments are assigned approximately the same number of active transactions. Distribution does *not* depend on the size of the available rollback segments. Therefore, in environments where all transactions generate the same amount of rollback information, all rollback segments can be the same size.

> **Note:** The number of transactions that a rollback segment can handle is a function of the data block size, which depends on the operating system.
>
> See your Oracle operating system specific documentation for more information.

When you create a rollback segment, you can specify storage parameters to control the allocation of extents for that segment. Each rollback segment must have at least two extents allocated.

One transaction writes sequentially to a single rollback segment. Each transaction writes to only one extent of the rollback segment at any given time. Many **active** transactions can write concurrently to a single rollback segment—even the same extent of a rollback segment. However, each data block in a rollback segment's extent can contain information for only a single transaction.

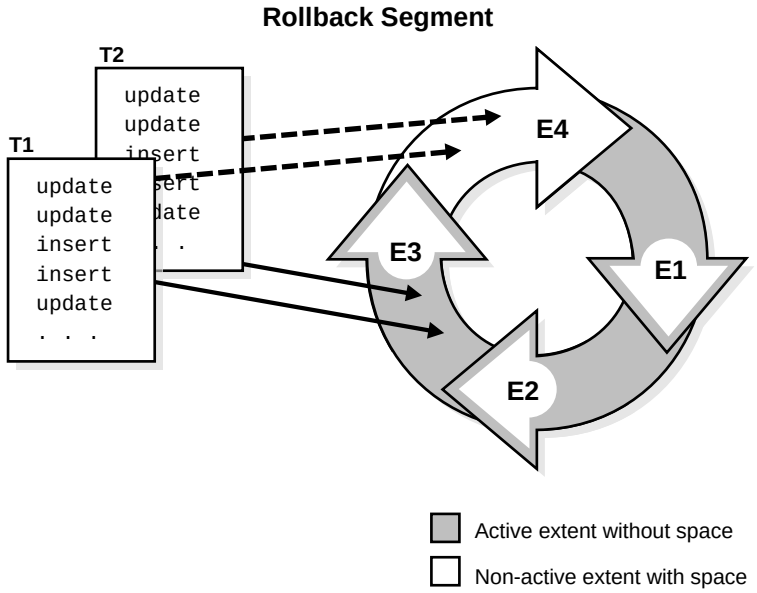
When a transaction runs out of space in the current extent and needs to continue writing, Oracle finds an available extent of the same rollback segment in one of two ways:

- It can reuse an extent already allocated to the rollback segment.
- It can acquire (and allocate) a new extent for the rollback segment.

The first transaction that needs to acquire more rollback space checks the next extent of the rollback segment. If the next extent of the rollback segment does not contain information from an active transaction, Oracle makes it the current extent, and all transactions that need more space from then on can write rollback information to the new current extent.

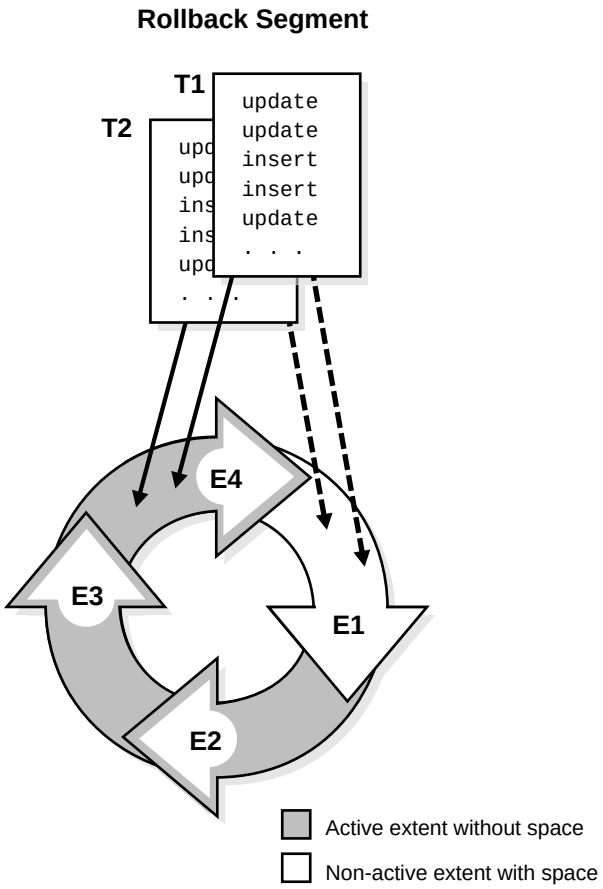
Figure 3–6 illustrates two transactions, T1 and T2, which begin writing in the third extent (E3) and continue writing to the fourth extent (E4) of a rollback segment.

Figure 3–6 Use of Allocated Extents in a Rollback Segment



As the transactions continue writing and fill the current extent, Oracle checks the next extent already allocated for the rollback segment to determine if it is available. In Figure 3–7, when E4 is completely full, T1 and T2 continue any further writing to the next extent allocated for the rollback segment that is available. In this figure, E1 is the next extent. This figure shows the cyclical nature of extent use in rollback segments.

Figure 3–7 Cyclical Use of the Allocated Extents in a Rollback Segment

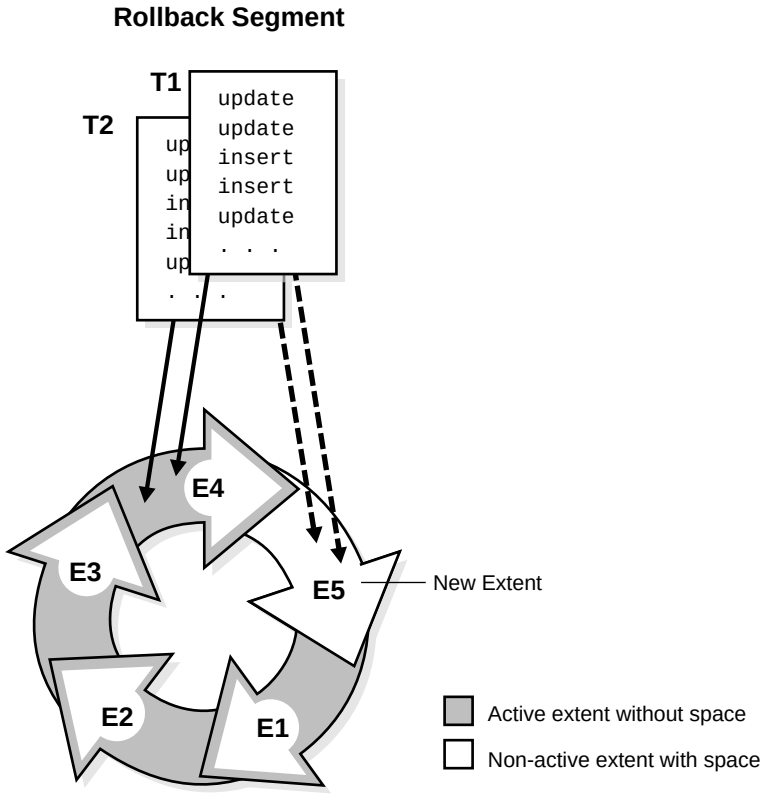


To continue writing rollback information for a transaction, Oracle always tries to reuse the next extent in the ring first. However, if the next extent contains data from active transaction, then Oracle must allocate a new extent. Oracle can allocate new extents for a rollback segment until the number of extents reaches the value set for the rollback segment's storage parameter `MAXEXTENTS`.

Figure 3–8 shows a new extent allocated for a rollback segment. The uncommitted transactions are long running (either idle, active, or persistent in-doubt distributed transactions). At this time, they are writing to the fourth extent, E4, in the rollback

segment. However, when E4 is completely full, the transactions cannot continue further writing to the next extent in sequence, E1, because it contains active rollback entries. Therefore, Oracle allocates a new extent, E5, for this rollback segment, and the transactions continue writing to this new extent.

Figure 3–8 Allocation of a New Extent for a Rollback Segment



How Extents Are Deallocated from a Rollback Segment

When you drop a rollback segment, Oracle returns all extents of the rollback segment to its tablespace. The returned extents are then available to other segments in the tablespace.

When you create or alter a rollback segment, you can use the storage parameter OPTIMAL (which applies *only* to rollback segments) to specify the optimal size of the segment in bytes. If a transaction needs to continue writing rollback information from one extent to another extent in the rollback segment, Oracle compares the current size of the rollback segment to the segment's optimal size. If the rollback segment is larger than its optimal size, and if the extents immediately following the extent just filled are inactive, then Oracle deallocates consecutive nonactive extents from the rollback segment until the total size of the rollback segment is equal to or close to, but not less than, its optimal size. Oracle always frees the oldest inactive extents, as these are the least likely to be used by consistent reads.

A rollback segment's OPTIMAL setting cannot be less than the combined space allocated for the minimum number of extents for the segment. For example:

(INITIAL + NEXT + NEXT + ... *up to MINEXTENTS*) bytes

The Rollback Segment SYSTEM

Oracle creates an initial rollback segment called SYSTEM whenever a database is created. This segment is in the SYSTEM tablespace and uses that tablespace's default storage parameters. You cannot drop the SYSTEM rollback segment. An instance always acquires the SYSTEM rollback segment in addition to any other rollback segments it needs.

If there are multiple rollback segments, Oracle tries to use the SYSTEM rollback segment only for special system transactions and distributes user transactions among other rollback segments. If there are too many transactions for the nonSYSTEM rollback segments, Oracle uses the SYSTEM segment as necessary. In general, after database creation, create at least one additional rollback segment in the SYSTEM tablespace.

Oracle Instances and Types of Rollback Segments

When an Oracle instance opens a database, it must acquire one or more rollback segments so that the instance can handle rollback information produced by subsequent transactions. An instance can acquire both private and public rollback segments. A **private rollback segment** is acquired explicitly by an instance when the instance opens a database. **Public rollback segments** form a pool of rollback segments that any instance requiring a rollback segment can use.

Any number of private and public rollback segments can exist in a database. As an instance opens a database, the instance attempts to acquire one or more rollback segments according to the following rules:

1. The instance must acquire at least one rollback segment. If the instance is the only instance accessing the database, it acquires the SYSTEM segment. If the instance is one of several instances accessing the database in an Oracle9*i* Real Application Clusters environment, it acquires the SYSTEM rollback segment and at least one other rollback segment. If it cannot, Oracle returns an error, and the instance cannot open the database.
2. The instance always attempts to acquire at least the number of rollback segments equal to the quotient of the values for the following initialization parameters:

   ```
   CEIL(TRANSACTIONS/TRANSACTIONS_PER_ROLLBACK_SEGMENT)
   ```

   CEIL is a SQL function that returns the smallest integer greater than or equal to the numeric input. In the previous example, if TRANSACTIONS equal 155 and TRANSACTIONS_PER_ROLLBACK_SEGMENT equal 10, then the instance tries to acquire at least 16 rollback segments. (However, an instance can open the database even if the instance cannot acquire the number of rollback segments given by the division in the previous example.)

   Note: The TRANSACTIONS_PER_ROLLBACK_SEGMENT parameter does not limit the number of transactions that can use a rollback segment. Rather, it determines the number of rollback segments an instance attempts to acquire when opening a database.

3. After acquiring the SYSTEM rollback segment, the instance next tries to acquire all private rollback segments specified by the instance's ROLLBACK_SEGMENTS parameter. If one instance in Oracle9*i* Real Application Clusters opens a database and attempts to acquire a private rollback segment already claimed by another instance, the second instance trying to acquire the rollback segment receives an error during startup. An error is also returned if an instance attempts to acquire a private rollback segment that does not exist.
4. If the instance has acquired enough private rollback segments in number 3, no further action is required. However, if an instance requires more rollback segments, the instance attempts to acquire public rollback segments.

   Once an instance claims a public rollback segment, no other instance can use that segment until either the rollback segment is taken offline or the instance that claimed the rollback segment is shut down.

A database used by Oracle9*i* Real Application Clusters can have both public and no private segments. Use of private segments is recommended.

See Also:

- *Oracle9i Real Application Clusters Concepts*
- *Oracle9i Real Application Clusters Administration*

for more information about rollback segment use with Oracle9*i* Real Application Clusters

Rollback Segment States

A rollback segment is always in one of several states, depending on whether it is offline, acquired by an instance, involved in an unresolved transaction, in need of recovery, or dropped. The state of the rollback segment determines whether it can be used in transactions, as well as which administrative procedures a DBA can perform on it.

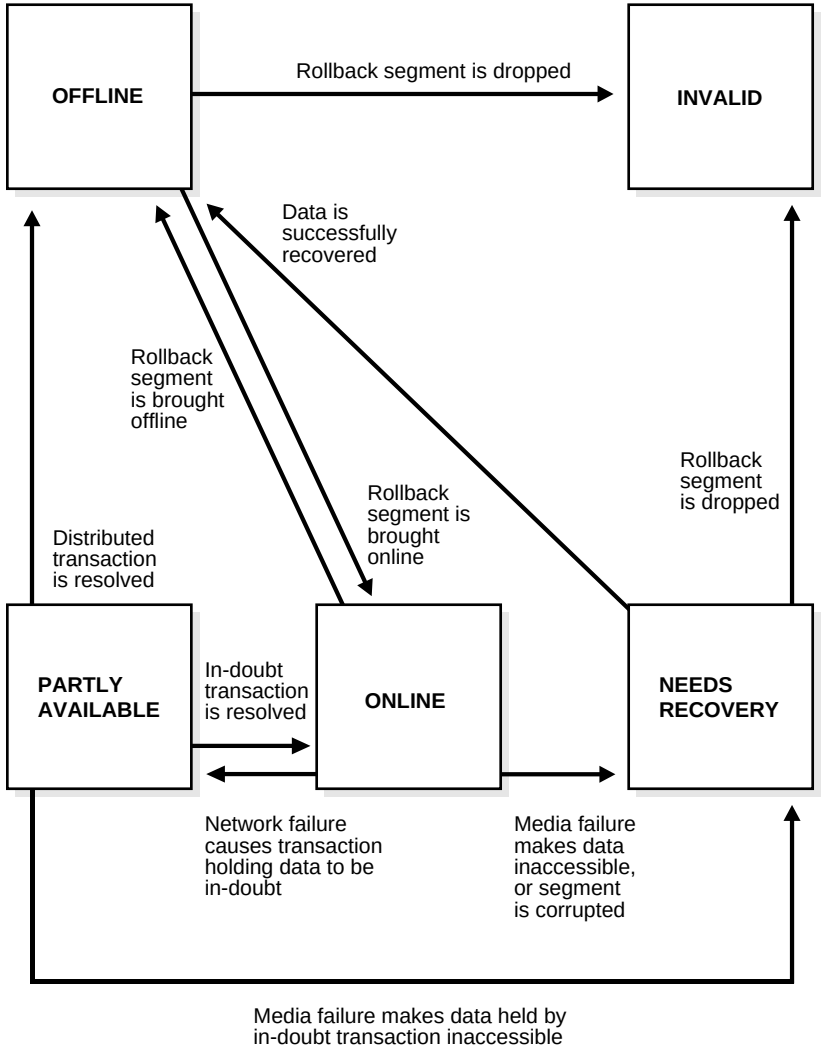
The rollback segment states are:

OFFLINE	Has not been acquired (brought online) by any instance.
ONLINE	Has been acquired (brought online) by an instance and can contain data from active transactions.
NEEDS RECOVERY	Contains data from uncommitted transactions that cannot be rolled back (because the data files involved are inaccessible) or is corrupted.
PARTLY AVAILABLE	Contains data from an in-doubt transaction (that is, an unresolved distributed transaction).
INVALID	Has been dropped (The space once allocated to this rollback segment will later be used when a new rollback segment is created.)

The data dictionary table `DBA_ROLLBACK_SEGS` lists the state of each rollback segment, along with other rollback information.

Figure 3–9 shows how a rollback segment moves from one state to another.

Figure 3–9 Rollback Segment States and State Transitions



PARTLY AVAILABLE and NEEDS RECOVERY Rollback Segments The `PARTLY AVAILABLE` and `NEEDS RECOVERY` states are very similar. A rollback segment in either state usually contains data from an unresolved transaction.

- A PARTLY AVAILABLE rollback segment is being used by an in-doubt distributed transaction that cannot be resolved because of a network failure. A NEEDS RECOVERY rollback segment is being used by a transaction (local or distributed) that cannot be resolved because of a local media failure, such as a missing or corrupted datafile, or is itself corrupted.
- Oracle or a DBA can bring a PARTLY AVAILABLE rollback segment online. In contrast, you must take a NEEDS RECOVERY rollback segment OFFLINE before it can be brought online. (If you recover the database and thereby resolve the transaction, Oracle automatically changes the state of the NEEDS RECOVERY rollback segment to OFFLINE.)
- A DBA can drop a NEEDS RECOVERY rollback segment. (This lets the DBA drop corrupted segments.) A PARTLY AVAILABLE segment cannot be dropped. You must first resolve the in-doubt transaction, either automatically by the RECO process or manually.

If you bring a PARTLY AVAILABLE rollback segment online (by a statement or during instance startup), Oracle can use it for new transactions. However, the in-doubt transaction still holds some of its transaction table entries, so the number of new transactions that can use the rollback segment is limited.

Until you resolve the in-doubt transaction, the transaction continues to hold the extents it acquired in the rollback segment, preventing other transactions from using them. Thus, the rollback segment may need to acquire new extents for the active transactions, and therefore grow. To prevent the rollback segment from growing, a database administrator can create a new rollback segment for transactions to use until the in-doubt transaction is resolved, rather than bring the PARTLY AVAILABLE segment online.

See Also:

- *Oracle9i Database Administrator's Guide* for information about failures in distributed transactions
- "When Rollback Information Is Required" on page 3-25 for information on the transaction table

Deferred Rollback Segments

When a tablespace goes offline so that transactions cannot be rolled back immediately, Oracle writes to a **deferred rollback segment**. The deferred rollback segment contains the rollback entries that could not be applied to the tablespace, so that they can be applied when the tablespace comes back online. These segments

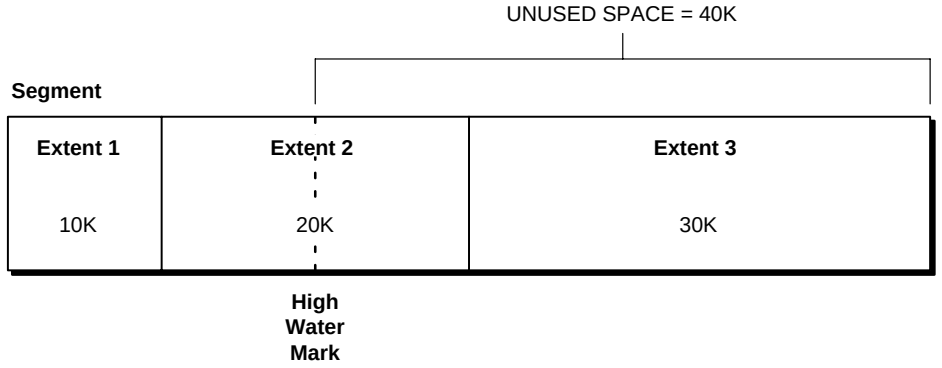
disappear as soon as the tablespace is brought back online and recovered. Oracle automatically creates deferred rollback segments in the `SYSTEM` tablespace.

High Water Mark

The **high water mark** is the boundary between used and unused space in a segment. As requests for new free blocks that cannot be satisfied by existing free lists are received, the block to which the high water mark points becomes a used block, and the high water mark is advanced to the next block. In other words, the segment space to the left of the high water mark is used, and the space to the right of it is unused.

Figure 3–10 shows a segment consisting of three extents containing 10K, 20K, and 30K of space, respectively. The high water mark is in the middle of the second extent. Thus, the segment contains 20K of used space to the left of the high water mark, and 40K of unused space to the right of the high water mark.

Figure 3–10 High Water Mark



4

Tablespaces, Datafiles, and Control Files

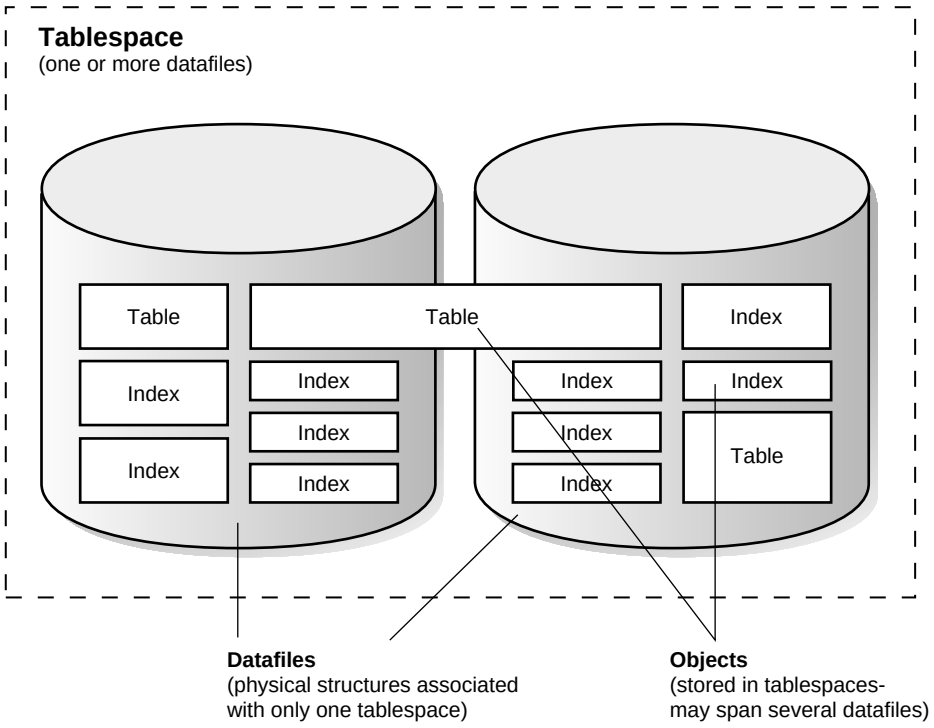
This chapter describes tablespaces, the primary logical database structures of any Oracle database, and the physical datafiles that correspond to each tablespace. The chapter includes:

- Introduction to Tablespaces, Datafiles, and Control Files
- Tablespaces Overview
- Datafiles Overview
- Control Files Overview

Introduction to Tablespaces, Datafiles, and Control Files

Oracle stores data logically in **tablespaces** and physically in **datafiles** associated with the corresponding tablespace. Figure 4–1 illustrates this relationship.

Figure 4–1 Datafiles and Tablespaces



Databases, tablespaces, and datafiles are closely related, but they have important differences:

Databases and tablespaces	An Oracle database consists of one or more logical storage units called tablespaces, which collectively store all of the database's data.
Tablespaces and datafiles	Each tablespace in an Oracle database consists of one or more files called datafiles, which are physical structures that conform with the operating system in which Oracle is running.

Databases and datafiles	A database's data is collectively stored in the datafiles that constitute each tablespace of the database. For example, the simplest Oracle database would have one tablespace and one datafile. Another database can have three tablespaces, each consisting of two datafiles (for a total of six datafiles).

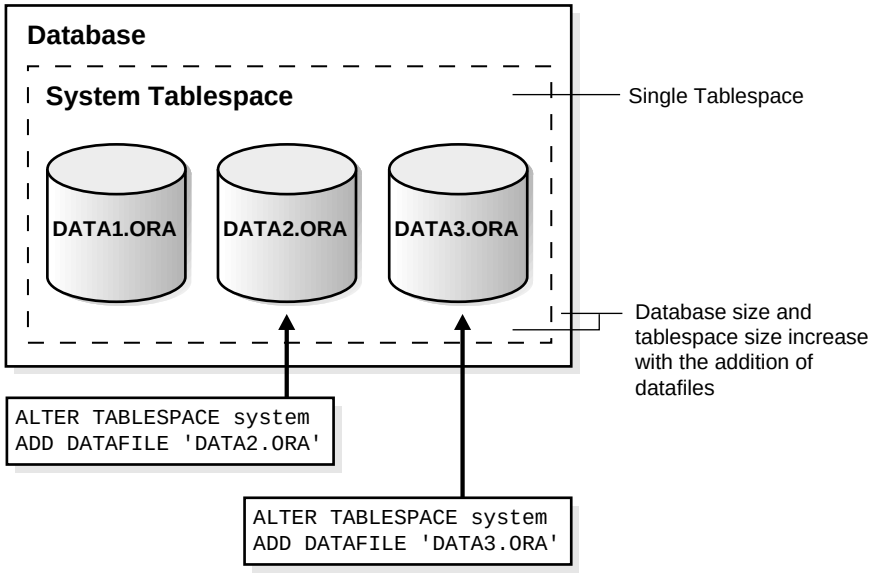
Allocate More Space for a Database

You can enlarge a database in three ways:

- Add a datafile to a tablespace
- Add a new tablespace
- Increase the size of a datafile

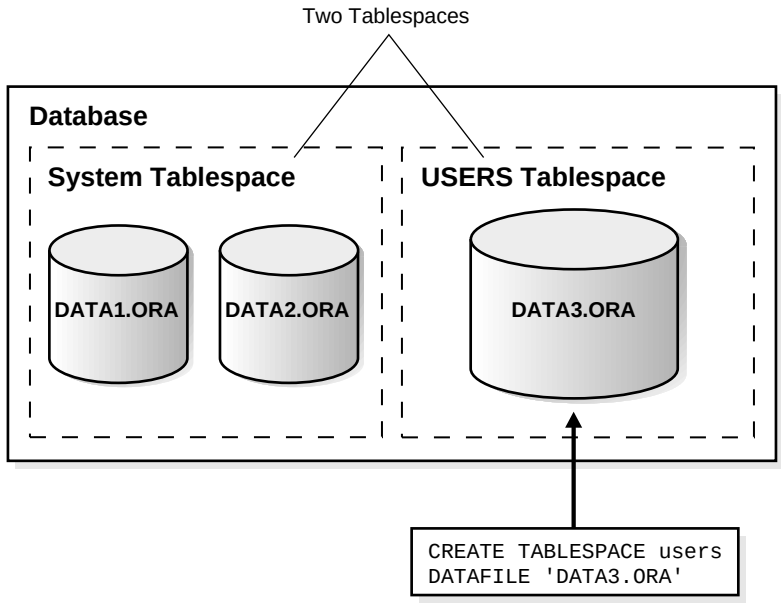
When you add another datafile to an existing tablespace, you increase the amount of disk space allocated for the corresponding tablespace. Figure 4–2 illustrates this kind of space increase.

Figure 4–2 Enlarging a Database by Adding a Datafile to a Tablespace



Alternatively, you can create a new tablespace (which contains at least one additional datafile) to increase the size of a database. Figure 4–3 illustrates this.

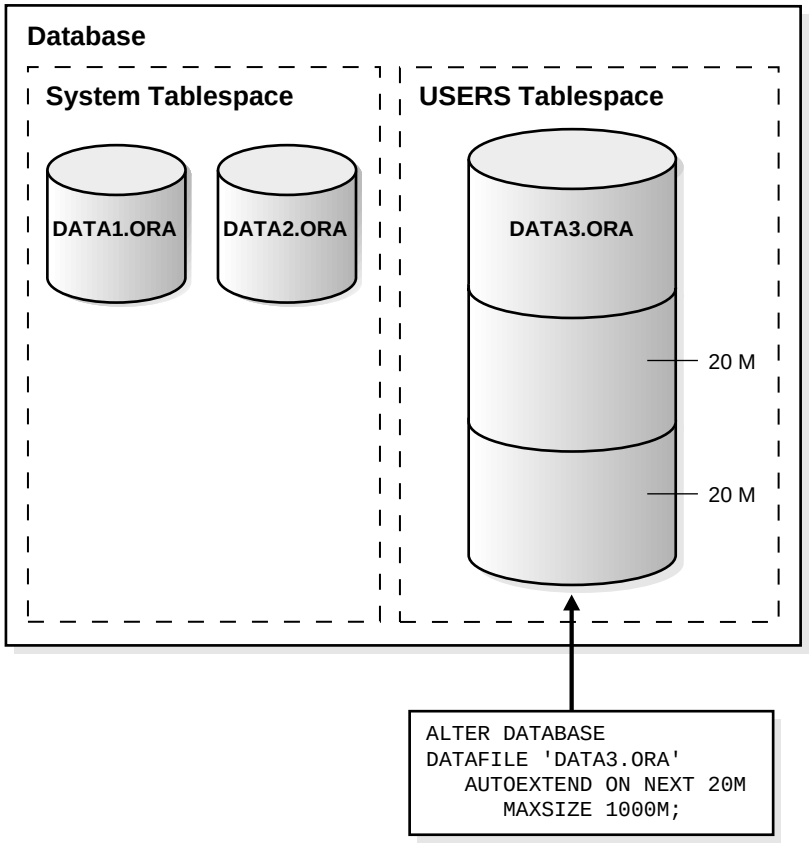
Figure 4–3 Enlarging a Database by Adding a New Tablespace



The size of a tablespace is the size of the datafiles that constitute the tablespace. The size of a database is the collective size of the tablespaces that constitute the database.

The third option for enlarging a database is to change a datafile's size or let datafiles in existing tablespaces grow dynamically as more space is needed. You accomplish this by altering existing files or by adding files with dynamic extension properties. Figure 4–4 illustrates this.

Figure 4–4 Enlarging a Database by Dynamically Sizing Datafiles



See Also: *Oracle9i Database Administrator's Guide* for more information about increasing the amount of space in your database

Tablespaces Overview

A database is divided into one or more logical storage units called tablespaces. Tablespaces are divided into logical units of storage called **segments**, which are further divided into **extents**. Extents are a collection of contiguous blocks.

This section includes the following topics about tablespaces:

- The SYSTEM Tablespace
- Undo Tablespaces
- Default Temporary Tablespace
- Multiple Tablespace Usage
- Space Management in Tablespaces
- Nonstandard Block Sizes
- Online and Offline Tablespaces
- Read-Only Tablespaces
- Temporary Tablespaces for Sorts
- Transport of Tablespaces between Databases

> **See Also:** Chapter 3, "Data Blocks, Extents, and Segments" for more information about segments and extents.

The SYSTEM Tablespace

Every Oracle database contains a tablespace named `SYSTEM`, which Oracle creates automatically when the database is created.

> **Note:** The `SYSTEM` tablespace is always online when the database is open.

The Data Dictionary

The `SYSTEM` tablespace always contains the data dictionary tables for the entire database. The data dictionary tables are stored in `datafile 1`.

PL/SQL Program Units Description

All data stored on behalf of stored PL/SQL program units (that is, procedures, functions, packages, and triggers) resides in the `SYSTEM` tablespace. If the database will contain many of these program units, the database administrator needs to provide the space they need in the `SYSTEM` tablespace.

See Also:

- "Online and Offline Tablespaces" on page 4-14 for more information about the permanent online condition of the `SYSTEM` tablespace
- Chapter 16, "SQL, PL/SQL, and Java" and Chapter 18, "Triggers" for more information about the space requirements of PL/SQL program units

Undo Tablespaces

Undo tablespaces are special tablespaces used solely for storing undo information. You cannot create any other types of segment (for example, tables or indexes) in undo tablespaces. Each database contains zero or more undo tablespaces. In automatic undo management mode, each Oracle instance is assigned one (and only one) undo tablespace. Undo data is managed within an undo tablespace using undo segments that are automatically created and maintained by Oracle.

When the first DML operation is executed within a transaction, the transaction is bound (assigned) to an undo segment (and therefore to a transaction table) in the current undo tablespace. In rare circumstances, if the instance does not have a designated undo tablespace, the transaction binds to the System Undo Segment.

Caution: Do not run any user transactions before creating and onlining the first undo tablespace.

Each undo tablespace is composed of a set of undo files and is locally managed. Like other types of tablespaces, undo blocks are grouped in extents and the status of each extent is represented in the bitmap. At any point in time, an extent is either allocated to (and used by) a transaction table, or it is free.

Creation of Undo Tablespaces

A DBA creates undo tablespaces individually, using the `CREATE UNDO TABLESPACE` statement. It can also be created when the database is created, using

the CREATE DATABASE statement. A set of files is assigned to each newly created undo tablespace. Like regular tablespaces, attributes of undo tablespaces can be modified with the ALTER TABLESPACE statement and dropped with the DROP TABLESPACE statement.

> **Note:** An undo tablespace cannot be dropped if it is being used by any instance or contains any undo information needed to recover transactions.

Assignment of Undo Tablespaces

You assign an undo tablespace to an instance in one of two ways:

- At instance startup. You can specify the undo tablespace in the initialization file or let the system choose an available undo tablespace.
- While the instance is running. Use ALTER SYSTEM SET UNDO_TABLESPACE to replacing the active undo tablespace with another undo tablespace. This method is rarely used.

You can add more space to an undo tablespace by adding more data files to the undo tablespace with the ALTER TABLESPACE statement.

You can have more than one undo tablespace and switch between them. You can use the Database Resource Manager to establish user quotas for undo tablespaces. You can specify the retention period for undo information.

> **See Also:** *Oracle9i Database Administrator's Guide* for detailed information about creating and managing undo tablespaces

Default Temporary Tablespace

If you have not created a temporary tablespace for a user, Oracle must have somewhere to store temporary data for that user. Historically, Oracle has used SYSTEM for default temporary data storage.

In Oracle9*i*, Release 1 (9.0.1), you are encouraged to define a default temporary tablespace when creating the database. If you do not, SYSTEM will still be used for default temporary storage. However, you will receive a warning in alert.log saying that a default temporary tablespace is recommended and will be necessary in future releases.

Note: Future releases of Oracle will enable creation of databases that are managed entirely locally. When SYSTEM is permanent and locally managed, it cannot be used for default temporary storage. Users will then be *required* to define a default temporary tablespace when creating a database.

How to Specify a Default Temporary Tablespace

You specify a default temporary tablespace when you create a database, using the DEFAULT TEMPORARY TABLESPACE extension to the CREATE DATABASE statement.

You can drop the default temporary tablespace. If you do, the SYSTEM tablespace will be used as default temporary tablespace. However, in future releases, this might not be allowed.

Note: You cannot make the default temporary tablespace permanent or take it offline.

See Also: *Oracle9i SQL Reference* for more information about defining and altering default temporary tablespaces

Multiple Tablespace Usage

A very small database may need only the SYSTEM tablespace; however, Oracle Corporation recommends that you create at least one additional tablespace to store user data separate from data dictionary information. This gives you more flexibility in various database administration operations and reduces contention among dictionary objects and schema objects for the same datafiles.

You can use multiple tablespaces to:

- Control disk space allocation for database data
- Assign specific space quotas for database users
- Control availability of data by taking individual tablespaces online or offline
- Perform partial database backup or recovery operations
- Allocate data storage across devices to improve performance

A database administrator (DBA) can do the following:

- Create new tablespaces
- Add datafiles to tablespaces
- Set and alter default segment storage settings for segments created in a tablespace
- Make a tablespace read-only or read / write
- Make a tablespace temporary or permanent
- Drop tablespaces.

Space Management in Tablespaces

Tablespaces allocate space in extents. Tablespaces can use two different methods to keep track of their free and used space:

- Extent management by the data dictionary *(***dictionary-managed tablespaces***)*
- Extent management by the tablespace *(***locally managed tablespaces***)*

When you create a tablespace, you choose one of these methods of space management. You cannot alter the method at a later time.

Note: With Oracle9*i*, Release 1 (9.0.1), if you do not specify extent management when you create a tablespace, the default is locally managed. In prior versions, the default was dictionary-managed.

See Also: "Extents Overview" on page 3-13

Dictionary-Managed Tablespaces

For a tablespace that uses the data dictionary to manage its extents, Oracle updates the appropriate tables in the data dictionary whenever an extent is allocated or freed for reuse. Oracle also stores rollback information about each update of the dictionary tables. Because dictionary tables and rollback segments are part of the database, the space that they occupy is subject to the same space management operations as all other data.

See Also: "Introduction to Rollback Segments" on page 3-24 for information about the storage of rollback information about dictionary tables

Locally Managed Tablespaces

A tablespace that manages its own extents maintains a bitmap in each datafile to keep track of the free or used status of blocks in that datafile. Each bit in the bitmap corresponds to a block or a group of blocks. When an extent is allocated or freed for reuse, Oracle changes the bitmap values to show the new status of the blocks. These changes do not generate rollback information because they do not update tables in the data dictionary (except for special cases such as tablespace quota information).

Locally managed tablespaces have the following advantages over dictionary-managed tablespaces:

- Local management of extents avoids recursive space management operations, which 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 rollback segment or data dictionary table.
- Local management of extents automatically tracks adjacent free space, eliminating the need to coalesce free extents.

The sizes of extents that are managed locally can be determined automatically by the system. Alternatively, all extents can have the same size in a locally managed tablespace and will override object storage options.

The `LOCAL` clause of the `CREATE TABLESPACE` or `CREATE TEMPORARY TABLESPACE` statement is specified to create locally managed permanent or temporary tablespaces, respectively.

Segment Space Management in Locally Managed Tablespaces

When you create a locally managed tablespace using the `CREATE TABLESPACE` statement, the `SEGMENT SPACE MANAGEMENT` clause lets you specify how free and used space within a segment is to be managed. Your choices are:

- `MANUAL`

  Specifying this keyword tells Oracle that you want to use free lists for managing free space within segments. Free lists are lists of data blocks that have space available for inserting rows. `MANUAL` is the default.

- `AUTO`

  This keyword tells Oracle that you want to use bitmaps to manage the free space within segments. A bitmap, in this case, is a map that describes the status of each data block within a segment with respect to the amount of space in the block available for inserting rows. As more or less space becomes available in a data block, its new state is reflected in the bitmap. Bitmaps enable Oracle to

manage free space more automatically, and thus, this form of space management is called automatic segment-space management.

See Also:

- *Oracle9i SQL Reference* for details about SQL statements
- *Oracle9i Database Administrator's Guide* for more information about managing SQL statements
- "Extents Managed Locally" on page 3-14 for more information about extent size
- "Temporary Tablespaces for Sorts" on page 4-16 for more information about temporary tablespaces

Nonstandard Block Sizes

The block size of the `SYSTEM` tablespace is termed the **standard block size**. This is set when the database is created and can be any valid size.

Oracle9*i*, Release 1 (9.0.1), lets you specify up to four nonstandard block sizes, in addition to a standard block size. In the initialization file, you can configure subcaches within the buffer cache for each of these block sizes. Subcaches can also be configured while an instance is running. You can create tablespaces having any of these block sizes. The standard block size is used for the system tablespace and most other tablespaces.

Note: All partitions of a partitioned object must reside in tablespaces of a single block size.

Multiple block sizes are useful primarily when transporting a tablespace from an OLTP database to an enterprise data warehouse. Oracle9*i*, Release 1 (9.0.1), facilitates transport between databases of different block sizes.

See Also:

- "Transport of Tablespaces between Databases" on page 4-17
- *Oracle9i Data Warehousing Guide* for information about transporting tablespaces in data warehousing environments

Online and Offline Tablespaces

A database administrator can bring any tablespace other than the SYSTEM tablespace **online** (accessible) or **offline** (not accessible) whenever the database is open. The SYSTEM tablespace is always online when the database is open because the data dictionary must always be available to Oracle.

A tablespace is normally online so that the data contained within it is available to database users. However, the database administrator can take a tablespace offline for maintenance or backup and recovery purposes:

When a Tablespace Goes Offline

When a tablespace goes offline, Oracle does not permit any subsequent SQL statements to reference objects contained in that tablespace. Active transactions with completed statements that refer to data in that tablespace are not affected at the transaction level. Oracle saves rollback data corresponding to those completed statements in a deferred rollback segment in the SYSTEM tablespace. When the tablespace is brought back online, Oracle applies the rollback data to the tablespace, if needed.

When a tablespace goes offline or comes back online, this is recorded in the data dictionary in the SYSTEM tablespace. If a tablespace is offline when you shut down a database, the tablespace remains offline when the database is subsequently mounted and reopened.

You can bring a tablespace online only in the database in which it was created because the necessary data dictionary information is maintained in the SYSTEM tablespace of that database. An offline tablespace cannot be read or edited by any utility other than Oracle. Thus, offline tablespaces cannot be transposed to other databases.

Oracle automatically switches a tablespace from online to offline when certain errors are encountered. For example, Oracle switches a tablespace from online to offline when the database writer process, DBW*n*, fails in several attempts to write to a datafile of the tablespace. Users trying to access tables in the offline tablespace receive an error. If the problem that causes this disk I/O to fail is media failure, you must recover the tablespace after you correct the hardware problem.

See Also:

- "Temporary Tablespaces for Sorts" on page 4-16 for more information about transferring online tablespaces between databases
- *Oracle9i Database Utilities* for more information about tools for data transfer

Use of Tablespaces for Special Procedures

If you create multiple tablespaces to separate different types of data, you take specific tablespaces offline for various procedures. Other tablespaces remain online, and the information in them is still available for use. However, special circumstances can occur when tablespaces are taken offline. For example, if two tablespaces are used to separate table data from index data, the following is true:

- If the tablespace containing the indexes is offline, queries can still access table data because queries do not require an index to access the table data.
- If the tablespace containing the tables is offline, the table data in the database is not accessible because the tables are required to access the data.

In summary, if Oracle has enough information in the online tablespaces to execute a statement, it does so. If it needs data in an offline tablespace, then it causes the statement to fail.

Read-Only Tablespaces

The primary purpose of read-only tablespaces is to eliminate the need to perform backup and recovery of large, static portions of a database. Oracle never updates the files of a read-only tablespace, and therefore the files can reside on read-only media such as CD ROMs or WORM drives.

Note: Because you can only bring a tablespace online in the database in which it was created, read-only tablespaces are not meant to satisfy archiving or data publishing requirements.

Read-only tablespaces cannot be modified. To update a read-only tablespace, first make the tablespace read / write. After updating the tablespace, you can then reset it to be read-only.

Because read-only tablespaces cannot be modified, they do not need repeated backup. Also, if you need to recover your database, you do not need to recover any read-only tablespaces, because they could not have been modified.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about changing a tablespace to read-only or read/write mode
- *Oracle9i SQL Reference* for more information about the `ALTER TABLESPACE` statement
- *Oracle9i Backup and Recovery Concepts* for more information about recovery

Temporary Tablespaces for Sorts

You can manage space for sort operations more efficiently by designating **temporary tablespaces** exclusively for sorts. Doing so effectively eliminates serialization of space management operations involved in the allocation and deallocation of sort space.

All operations that use sorts—including joins, index builds, ordering (`ORDER BY`), the computation of aggregates (`GROUP BY`), and the `ANALYZE` statement for collecting optimizer statistics—benefit from temporary tablespaces. The performance gains are significant with Oracle9*i* Real Application Clusters.

Sort Segments

A temporary tablespace can be used only for sort segments. A temporary tablespace is not the same as a tablespace that a user designates for temporary segments, which can be any tablespace available to the user. No permanent schema objects can reside in a temporary tablespace.

Sort segments are used when a segment is shared by multiple sort operations. One sort segment exists for every instance that performs a sort operation in a given tablespace.

Temporary tablespaces provide performance improvements when you have multiple sorts that are too large to fit into memory. The sort segment of a given temporary tablespace is created at the time of the first sort operation. The sort segment expands by allocating extents until the segment size is equal to or greater than the total storage demands of all of the active sorts running on that instance.

See Also: Chapter 3, "Data Blocks, Extents, and Segments" for more information about segments

Creation of Temporary Tablespaces

You can create temporary tablespaces by using the `CREATE TABLESPACE` or `CREATE TEMPORARY TABLESPACE` statement.

See Also:

- "Temporary Datafiles" on page 4-20 for information about `TEMPFILES`
- "Space Management in Tablespaces" on page 4-11 for information about locally managed and dictionary-managed tablespaces
- *Oracle9i SQL Reference* for information about the `CREATE TABLESPACE`, `CREATE TEMPORARY TABLESPACE`, and `ALTER TABLESPACE` statements
- *Oracle9i Database Performance Guide and Reference* for information about setting up temporary tablespaces for sorts and hash joins

Transport of Tablespaces between Databases

The **transportable tablespace** feature enables you to move a subset of an Oracle database from one Oracle database to another on the same platform. You can clone a tablespace from one tablespace and plug it into another database, copying the tablespace between databases, or you can unplug a tablespace from one Oracle database and plug it into another Oracle database, moving the tablespace between databases on the same platform.

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 datafiles and integrating the tablespace metadata. When you transport tablespaces you can also move index data, so you do not have to rebuild the indexes after importing or loading the table data.

Note: You can transport tablespaces only between Oracle databases that use the same character set and that run on compatible platforms from the same hardware vendor.

How to Move or Copy a Tablespace to Another Database

To move or copy a set of tablespaces, you must make the tablespaces read-only, copy the datafiles of these tablespaces, and use export/import to move the database information *(***metadata***)* stored in the data dictionary. Both the datafiles and the metadata export file must be copied to the target database. The transport of these files can be done using any facility for copying flat files, such as the operating system copying facility, ftp, or publishing on CDs.

After copying the datafiles and importing the metadata, you can optionally put the tablespaces in read/write mode.

See Also:

- *Oracle9i Database Administrator's Guide* for details about how to move or copy tablespaces to another database
- *Oracle9i Database Utilities* for import/export information

Datafiles Overview

A tablespace in an Oracle database consists of one or more physical **datafiles**. A datafile can be associated with only one tablespace and only one database.

Oracle creates a datafile for a tablespace by allocating the specified amount of disk space plus the overhead required for the file header. When a datafile is created, the operating system in which Oracle is running is responsible for clearing old information and authorizations from a file before allocating it to Oracle. If the file is large, this process can take a significant amount of time. The first tablespace in any database is always the `SYSTEM` tablespace, so Oracle automatically allocates the first datafiles of any database for the `SYSTEM` tablespace during database creation.

See Also: Your Oracle operating system specific documentation for information about the amount of space required for the file header of datafiles on your operating system

Datafile Contents

When a datafile is first created, the allocated disk space is formatted but does not contain any user data. However, Oracle reserves the space to hold the data for future segments of the associated tablespace—it is used exclusively by Oracle. As the data grows in a tablespace, Oracle uses the free space in the associated datafiles to allocate extents for the segment.

The data associated with schema objects in a tablespace is physically stored in one or more of the datafiles that constitute the tablespace. Note that a schema object does not correspond to a specific datafile; rather, a datafile is a repository for the data of any schema object within a specific tablespace. Oracle allocates space for the data associated with a schema object in one or more datafiles of a tablespace. Therefore, a schema object can span one or more datafiles. Unless table **striping** is used (where data is spread across more than one disk), the database administrator and end users cannot control which datafile stores a schema object.

> **See Also:** Chapter 3, "Data Blocks, Extents, and Segments" for more information about use of space

Size of Datafiles

You can alter the size of a datafile after its creation or you can specify that a datafile should dynamically grow as schema objects in the tablespace grow. This functionality enables you to have fewer datafiles for each tablespace and can simplify administration of datafiles.

> **Note:** Make sure there is sufficient space on the operating system for expansion.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about resizing datafiles

Offline Datafiles

You can take tablespaces offline or bring them online at any time, except for the `SYSTEM` tablespace. All of the datafiles making up a tablespace are taken offline or brought online as a unit when you take the tablespace offline or bring it online, respectively.

You can take individual datafiles offline. However, this is normally done only during some database recovery procedures.

Temporary Datafiles

Locally managed temporary tablespaces have temporary datafiles (**tempfiles**), which are similar to ordinary datafiles except that:

- Tempfiles are always set to NOLOGGING mode.
- You cannot make a tempfile read-only.
- You cannot rename a tempfile.
- You cannot create a tempfile with the `ALTER DATABASE` statement.
- When you create or resize tempfiles, they are not always guaranteed allocation of disk space for the file size specified. On certain file systems (for example, UNIX) disk blocks are allocated not at file creation or resizing, but before the blocks are accessed.

  Caution: This enables fast tempfile creation and resizing; however, the disk could run of space later when the tempfiles are accessed.

- Tempfile information is shown in the 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: "Space Management in Tablespaces" on page 4-11 for more information about locally managed tablespaces

Control Files Overview

The control file of a database is a small binary file necessary for the database to start and operate successfully. A control file is updated continuously by Oracle during database use, so it must be available for writing whenever the database is open. If for some reason the control file is not accessible, the database will not function properly.

Each control file is associated with only one Oracle database.

Control File Contents

A control file contains information about the associated database that is required for the database to be accessed by an instance, both at startup and during normal

operation. A control file's information can be modified only by Oracle; no database administrator or end-user can edit a database's control file.

Among other things, a control file contains information such as:

- The database name
- The timestamp of database creation
- The names and locations of associated datafiles and online redo log files
- Tablespace information
- Datafile offline ranges
- The log history
- Archived log information
- Backup set and backup piece information
- Backup datafile and redo log information
- Datafile copy information
- The current log sequence number
- Checkpoint information

The database name and timestamp originate at database creation. The database's name is taken from either the name specified by the initialization parameter `DB_NAME` or the name used in the `CREATE DATABASE` statement.

Each time that a datafile or an online redo log file is added to, renamed in, or dropped from the database, the control file is updated to reflect this physical structure change. These changes are recorded so that:

- Oracle can identify the datafiles and online redo log files to open during database startup
- Oracle can identify files that are required or available in case database recovery is necessary

Therefore, if you make a change to your database's physical structure (use `ALTER DATABASE` commands), you should immediately make a backup of your control file.

Control files also record information about checkpoints. Every three seconds, the checkpoint process (CKPT) records information in the control file about the checkpoint position in the online redo log. This information is used during database recovery to tell Oracle that all redo entries recorded before this point in the online

redo log group are not necessary for database recovery; they were already written to the datafiles.

See Also:

- *Oracle9i Recovery Manager User's Guide*
- *Oracle9i User-Managed Backup and Recovery Guide*

for information about backing up a database's control file

Multiplexed Control Files

As with online redo log files, Oracle enables multiple, identical control files to be open concurrently and written for the same database.

By storing multiple control files for a single database on different disks, you can safeguard against a single point of failure with respect to control files. If a single disk that contained a control file crashes, the current instance fails when Oracle attempts to access the damaged control file. However, other copies of the current control file are available on different disks, so an instance can be restarted easily without the need for database recovery.

The permanent loss of all copies of a database's control file is a serious problem to safeguard against. If *all* control files of a database are permanently lost during operation (several disks fail), the instance is aborted and media recovery is required. Even so, media recovery is not straightforward if an older backup of a control file must be used because a current copy is not available. Therefore, it is strongly recommended that you adhere to the following practices:

- Use multiplexed control files with each database
- Store each copy on a different physical disk
- Use operating system mirroring
- Monitor backups

5

The Data Dictionary

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 includes:

- Introduction to the Data Dictionary
- How the Data Dictionary Is Used
- The Dynamic Performance Tables
- Database Object Metadata

Introduction to the Data Dictionary

One of the most important parts of an Oracle database is its **data dictionary**, which is a **read-only** set of tables that provides information about the database. A data dictionary contains:

- The definitions of all schema objects in the database (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- How much space has been allocated for, and is currently used by, the schema objects
- Default values for columns
- Integrity constraint information
- The names of Oracle users
- Privileges and roles each user has been granted
- Auditing information, such as who has accessed or updated various schema objects
- Other general database information

The data dictionary is structured in tables and views, just like other database data. All the data dictionary tables and views for a given database are stored in that database's `SYSTEM` tablespace.

Not only is the data dictionary central to every Oracle database, it is an important tool for all users, from end users to application designers and database administrators. To access the data dictionary, use SQL statements. Because the data dictionary is read-only, you can issue only queries (`SELECT` statements) against the tables and views of the data dictionary.

> **See Also:** "The SYSTEM Tablespace" on page 4-7 for more information about `SYSTEM` tablespaces

The Structure of the Data Dictionary

A database's data dictionary consists of:

Base tables	The underlying tables that store information about the associated database. Only Oracle should write to and read these tables. Users rarely access them directly because they are normalized, and most of the data is stored in a cryptic format.
User-accessible views	The views that summarize and display the information stored in the base tables of the data dictionary. 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. Most users are given access to the views rather than the base tables.

SYS, the Owner of the Data Dictionary

The Oracle user `SYS` owns all base tables and user-accessible views of the data dictionary. No Oracle user should *ever* alter (update, delete, or insert) any 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 underlying data dictionary tables can permanently and detrimentally affect the operation of a database.

How the Data Dictionary Is Used

The data dictionary has three primary uses:

- Oracle accesses the data dictionary to find information about users, schema objects, and storage structures.
- Oracle modifies the data dictionary every time that a data definition language (DDL) statement is issued.
- Any Oracle user can use the data dictionary as a read-only reference for information about the database.

How Oracle Uses the Data Dictionary

Data in the base tables of the data dictionary *is necessary for Oracle to function***.** Therefore, only Oracle should write or change data dictionary information. Oracle provides scripts to modify the data dictionary tables when a database is upgraded or downgraded.

> **Caution:** No data in any data dictionary table should be altered or deleted by any user.

During database operation, Oracle reads the data dictionary to ascertain that schema objects exist and that users have proper access to them. Oracle also updates the data dictionary continuously to reflect changes in database structures, auditing, grants, and data.

For example, if user KATHY creates a table named PARTS, new rows are added to the data dictionary that reflect the new table, columns, segment, extents, and the privileges that KATHY has on the table. This new information is then visible the next time the dictionary views are queried.

Public Synonyms for Data Dictionary Views

Oracle creates public synonyms on 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.

Cache the Data Dictionary for Fast Access

Much of the data dictionary information is cached in the SGA in the **dictionary cache**, because Oracle constantly accesses the data dictionary during database operation to validate user access and to verify the state of schema objects. All information is stored in memory using the LRU (least recently used) algorithm.

Information typically kept in the caches is that required for parsing. The COMMENTS columns describing the tables and their columns are not cached unless they are accessed frequently.

Other Programs and the Data Dictionary

Other Oracle 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: the synonyms are less likely to change between software releases.

How Users and DBAs Can Use the Data Dictionary

The views of the data dictionary serve as a reference for all database users. You access the data dictionary views through the SQL language. Some views are accessible to all Oracle users, and others are intended for database administrators only.

The data dictionary is always available when the database is open. It resides in the `SYSTEM` tablespace, which is always online.

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes:

Table 5–1 Data Dictionary View Prefixes

Prefix	Scope
`USER`	user's view (what is in the user's schema)
`ALL`	expanded user's view (what the user can access)
`DBA`	database administrator's view (what is in all users' schemas)

The set of columns is identical across views with these exceptions:

- 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.
- Some `DBA` views have additional columns containing information useful to the administrator.

> **See Also:** *Oracle9i Database Reference* for a complete list of data dictionary views and their columns

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 own private environment in the database, including information about schema objects created by the user, grants made by the user, and so on

- Display only rows pertinent to the user
- Have columns identical to the other views, except that the column OWNER is implied (the current user)
- Return a subset of the information in the ALL views
- 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;
```

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;
```

Views with the Prefix DBA

Views with the prefix DBA show a global view of the entire database. Therefore, they are meant to be queried only by database administrators. Any user granted the system privilege SELECT ANY TABLE can query the DBA-prefixed views of the data dictionary.

Synonyms are not created for these views, because the DBA views should be queried only by administrators. Therefore, to query the DBA views, administrators must prefix the view name with its owner, SYS, as in the following:

```
SELECT owner, object_name, object_type FROM SYS.DBA_OBJECTS;
```

Administrators can run the script file DBA_SYNONYMS.SQL to create private synonyms for the DBA views in their accounts if they have the SELECT ANY TABLE system privilege. Executing this script creates synonyms for the current user only.

The DUAL Table

The table named DUAL is a small table in the data dictionary that Oracle and user-written programs can reference to guarantee a known result. This table has one column called DUMMY and one row containing the value X.

See Also: *Oracle9i SQL Reference* for more information about the DUAL table

The Dynamic Performance Tables

Throughout its operation, Oracle maintains a set of "virtual" tables that record current database activity. These tables are called **dynamic performance tables**.

Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. These views are sometimes called **fixed views** because they cannot be altered or removed by the database administrator.

SYS owns the dynamic performance tables; their names all begin with V_$. Views are created on these tables, and then public synonyms are created for the views. The synonym names begin with V$. For example, the V$DATAFILE view contains information about the database's datafiles, and the V$FIXED_TABLE view contains information about all of the dynamic performance tables and views in the database.

See Also: *Oracle9i Database Reference* for a complete list of the dynamic performance views' synonyms and their columns

Database Object Metadata

Oracle9*i*, Release 1 (9.0.1), includes a PL/SQL package, DBMS_METADATA, which 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
- A simplified interface for ad hoc querying

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information about DBMS_METADATA

Part III

The Oracle Instance

Part III describes the architecture of the Oracle instance and explains the different client/server configurations it can have in a network environment. Part III also explains the Oracle startup and shutdown procedures.

Part III contains the following chapters:

- Chapter 6, "Database and Instance Startup and Shutdown"
- Chapter 7, "Distributed Processing"
- Chapter 8, "Memory Architecture"
- Chapter 9, "Process Architecture"
- Chapter 10, "Database Resource Management"

6

Database and Instance Startup and Shutdown

This chapter explains the procedures involved in starting and stopping an Oracle instance and database. It includes:

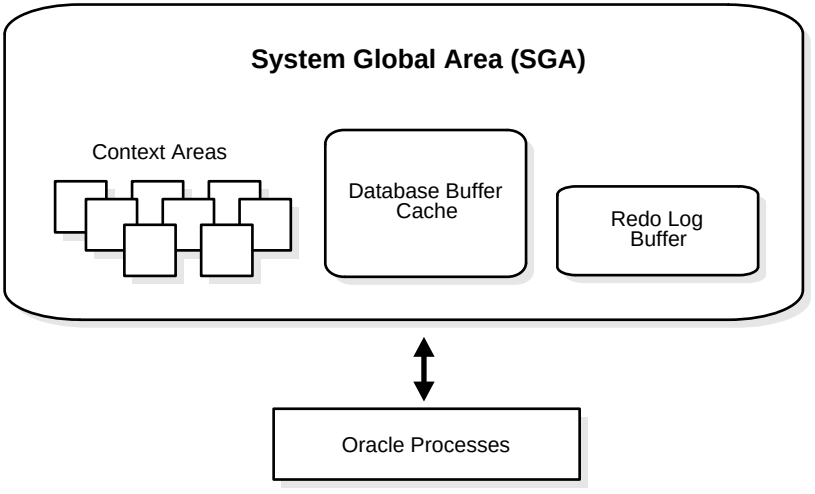
- Introduction to an Oracle Instance
- Instance and Database Startup
- Database and Instance Shutdown

Introduction to an Oracle Instance

Every running Oracle database is associated with an Oracle instance. When a database is started on a database server (regardless of the type of computer), Oracle allocates a memory area called the System Global Area (SGA) and starts one or more Oracle processes. This combination of the SGA and the Oracle processes is called an **Oracle instance**. The memory and processes of an instance manage the associated database's data efficiently and serve the one or multiple users of the database.

Figure 6–1 shows an Oracle instance.

Figure 6–1 An Oracle Instance



See Also:

- Chapter 8, "Memory Architecture"
- Chapter 9, "Process Architecture"

The Instance and the Database

After starting an instance, Oracle associates the instance with the specified database. This is called **mounting** the database. The database is then ready to be **opened**, which makes it accessible to authorized users.

Multiple instances can execute concurrently on the same computer, each accessing its own physical database. In clustered and massively parallel systems (MPP), Oracle9*i* Real Application Clusters enables multiple instances to mount a single database.

Only the database administrator can start up an instance and open the database. If a database is open, the database administrator can shut down the database so that it is closed. When a database is **closed**, users cannot access the information that it contains.

Security for database startup and shutdown is controlled through connections to Oracle with administrator privileges. Normal users do not have control over the current status of an Oracle database.

> **See Also:** *Oracle9i Real Application Clusters Concepts* for information

Connection with Administrator Privileges

Database startup and shutdown are powerful administrative options and are restricted to users who connect to Oracle with administrator privileges. Depending on the operating system**,** one of the following conditions establishes administrator privileges for a user:

- The user's operating system privileges allow him or her to connect using administrator privileges.
- The user is granted the `SYSDBA` or `SYSOPER` privileges, and the database uses password files to authenticate database administrators.

When you connect with `SYSDBA` privileges, you are placed in the schema owned by `SYS`. When you connect as `SYSOPER`, you are placed in the public schema. `SYSOPER` privileges are a subset of `SYSDBA` privileges.

See Also:

- Your operating system specific Oracle documentation for more information about how administrator privileges work on your operating system
- Chapter 24, "Controlling Database Access" for more information about password files and authentication schemes for database administrators

Initialization Parameter Files

To start an instance, Oracle must read an **initialization parameter file**—a file containing a list of configuration parameters for that instance and database. Set these parameters to particular values to initialize many of the memory and process settings of an Oracle instance. Most initialization parameters belong to one of the following groups:

- Parameters that name things (such as files)
- Parameters that set limits (such as maximums)
- Parameters that affect capacity (such as the size of the SGA), which are called **variable parameters**

Among other things, the initialization parameters tell Oracle:

- The name of the database for which to start up an instance
- How much memory to use for memory structures in the SGA
- What to do with filled online redo log files
- The names and locations of the database's control files
- The names of private rollback segments or undo tablespaces in the database

An Example of a Parameter File

The following is an example of a typical parameter file:

```
DB_BLOCK_BUFFERS = 550
DB_NAME = ORA8PROD
DB_DOMAIN = US.ACME.COM
#
LICENSE_MAX_USERS = 64
#
CONTROL_FILES = filename1, filename2
```

```
#
LOG_ARCHIVE_DEST = c:\logarch
LOG_ARCHIVE_FORMAT = arch%S.ora
LOG_ARCHIVE_START = TRUE
LOG_BUFFER = 64512
LOG_CHECKPOINT_INTERVAL = 256000
UNDO_MANAGEMENT = TRUE
UNDO_TABLESPACE = ut_one
```

How Parameter Values Are Changed

The database administrator can adjust variable parameters to improve the performance of a database system. Exactly which parameters most affect a system is a function of numerous database characteristics and variables.

Some parameters can be changed dynamically by using the `ALTER SESSION` or `ALTER SYSTEM` statement while the instance is running. Unless you are using a server parameter file, changes made using the `ALTER SYSTEM` statement are only in effect for the current instance. You must manually update the text initialization parameter file, for the changes to be known the next time you start up an instance. When you use a server parameter file, you can update the parameters on disk, so that changes persist across database shutdown and startup.

See Also:

- *Oracle9i Database Administrator's Guide* for a discussion of initialization parameters and the use of a server parameter file
- *Oracle9i Database Reference* for descriptions of all initialization parameters
- "Dynamic SGA" on page 8-4 for information about parameters that affect the SGA

Globalization Support Parameters

Oracle treats string literals defined for Globalization Support parameters in the file as if they are in the database character set.

See Also: *Oracle9i Globalization Support Guide* for more information about Globalization Support

Instance and Database Startup

The three steps to starting an Oracle database and making it available for systemwide use are:

1. Start an instance.
2. Mount the database.
3. Open the database.

A database administrator can perform these steps using the SQL*Plus `STARTUP` command or Oracle Enterprise Manager.

See Also: *Oracle Enterprise Manager Administrator's Guide*

How an Instance Is Started

When Oracle starts an instance, first it reads the initialization parameter file to determine the values of initialization parameters and then it allocates an SGA—a shared area of memory used for database information—and creates background processes. At this point, no database is associated with these memory structures and processes.

See Also:

- Chapter 8, "Memory Architecture" for information about the SGA
- Chapter 9, "Process Architecture" for information about background processes

Restricted Mode of Instance Startup

You can start an instance in restricted mode (or later alter an existing instance to be in restricted mode). This restricts connections to only those users who have been granted the `RESTRICTED SESSION` system privilege.

Forced Startup in Abnormal Situations

In unusual circumstances, a previous instance might not have been shut down cleanly. For example, one of the instance's processes might not have terminated properly. In such situations, the database can return an error during normal instance startup. To resolve this problem, you must terminate all remnant Oracle processes of the previous instance before starting the new instance.

How a Database Is Mounted

The instance mounts a database to associate the database with that instance. To mount the database, the instance finds the database control files and opens them. Control files are specified in the CONTROL_FILES initialization parameter in the parameter file used to start the instance. Oracle then reads the control files to get the names of the database's datafiles and redo log files.

At this point, the database is still closed and is accessible only to the database administrator. The database administrator can keep the database closed while completing specific maintenance operations. However, the database is not yet available for normal operations.

How a Database Is Mounted with Oracle9*i* Real Application Clusters

Note: The features described in this section are available only if you have purchased Oracle9*i* Enterprise Edition with Real Application Clusters.

If Oracle allows multiple instances to mount the same database concurrently, the database administrator can use the initialization parameter CLUSTER_DATABASE to make the database available to multiple instances. The default value of the CLUSTER_DATABASE parameter is FALSE. Versions of Oracle that do not support Real Application Clusters only allow CLUSTER_DATABASE to be FALSE.

If CLUSTER_DATABASE is FALSE for the first instance that mounts a database, only that instance can mount the database. If CLUSTER_DATABASE is set to TRUE on the first instance, other instances can mount the database if their CLUSTER_DATABASE parameters are set to TRUE. The number of instances that can mount the database is subject to a predetermined maximum, which you can specify when creating the database.

See Also:

- *Oracle9i Real Application Clusters Concepts*
- *Oracle9i Real Application Clusters Installation and Configuration*
- *Oracle9i Real Application Clusters Administration*
- *Oracle9i Real Application Clusters Deployment and Performance*

for more information about the use of multiple instances with a single database

How a Standby Database Is Mounted

A **standby database** maintains a duplicate copy of your primary database and provides continued availability in the event of a disaster.

The standby database is constantly in recovery mode. To maintain your standby database, you must mount it in standby mode using the `ALTER DATABASE` statement and apply the archived redo logs that your primary database generates.

You can open a standby database in read-only mode to use it as a temporary reporting database. You cannot open a standby database in read/write mode.

See Also:

- *Oracle9i Data Guard Concepts and Administration*
- "Open a Database in Read-Only Mode" on page 6-10 for information about opening a standby database in read-only mode

How a Clone Database Is Mounted

A **clone database** is a specialized copy of a database that can be used for tablespace point-in-time recovery. When you perform tablespace point-in-time recovery, you mount the clone database and recover the tablespaces to the desired time, then export metadata from the clone to the primary database and copy the datafiles from the recovered tablespaces.

See Also:

- *Oracle9i Recovery Manager User's Guide*
- *Oracle9i User-Managed Backup and Recovery Guide*

for detailed information about clone databases and tablespace point-in-time recovery

What Happens When You Open a Database

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 opens the online datafiles and online redo log files. If a tablespace was offline when the database was previously shut down, the tablespace and its corresponding datafiles will still be offline when you reopen the database.

If any of the datafiles or redo log files are not present when you attempt to open the database, Oracle returns an error. You must perform recovery on a backup of any damaged or missing files before you can open the database.

> **See Also:** "Online and Offline Tablespaces" on page 4-14 for information about opening an offline tablespace

Crash Recovery

If the database was last closed abnormally, either because the database administrator aborted its instance or because of a power failure, Oracle automatically performs crash recovery when the database is reopened.

Undo Space Acquisition and Management

When you open the database, the instance attempts to acquire one or more rollback segments or undo tablespaces. You determine whether to operate in manual undo management mode or automatic undo management mode at instance startup using the `UNDO_MANAGEMENT` initialization parameter. The supported values are `{AUTO/MANUAL}`. If `AUTO`, the instance will be started in automatic undo management mode. The default value is `MANUAL`.

- If you use the rollback segment method of managing undo space, you are using manual undo management mode.

- If you use the undo tablespace method, you are using automatic undo management mode.

  See Also: "The Rollback Segment SYSTEM" on page 3-31 and "Automatic Undo Management" on page 3-22 for more information about managing undo space.

Resolution of In-Doubt Distributed Transaction

Occasionally a database closes abnormally with one or more distributed transactions **in doubt** (neither committed nor rolled back). When you reopen the database and crash recovery is complete, the RECO background process automatically, immediately, and consistently resolves any in-doubt distributed transactions.

See Also: *Oracle9i Database Administrator's Guide* for information about recovery from distributed transaction failures

Open a Database in Read-Only Mode

You can open any database in read-only mode to prevent its data contents from being modified by user transactions. Read-only mode restricts database access to read-only transactions, which cannot write to the datafiles or to the redo log files.

Disk writes to other files, such as control files, operating system audit trails, trace files, and alert files, can continue in read-only mode. Temporary tablespaces for sort operations are not affected by the database being open in read-only mode. However, you cannot take permanent tablespaces offline while a database is open in read-only mode. Job queues are not available in read-only mode.

Read-only mode does not restrict database recovery or operations that change the database's state without generating redo data. For example, in read-only mode:

- Datafiles can be taken offline and online
- Recovery of offline datafiles and tablespaces can be performed
- The control file remains available for updates about the state of the database

One useful application of read-only mode occurs when standby databases function as temporary reporting databases.

See Also: *Oracle9i Database Administrator's Guide* for information about how to open a database in read-only mode

Database and Instance Shutdown

The three steps to shutting down a database and its associated instance are:

1. Close the database.
2. Unmount the database.
3. Shut down the instance.

A database administrator can perform these steps using Oracle Enterprise Manager. Oracle automatically performs all three steps whenever an instance is shut down.

See Also: *Oracle Enterprise Manager Administrator's Guide*

Close a Database

When you close a database, Oracle writes all database data and recovery data in the SGA to the datafiles and redo log files, respectively. Next, Oracle closes all online datafiles and online redo log files. (Any offline datafiles of any offline tablespaces have been closed already. If you subsequently reopen the database, any tablespace that was offline and its datafiles remain offline and closed, respectively.) At this point, the database is closed and inaccessible for normal operations. The control files remain open after a database is closed but still mounted.

Close the Database by Aborting the Instance

In rare emergency situations, you can abort the instance of an open database to close and completely shut down the database instantaneously. This process is fast, because the operation of writing all data in the buffers of the SGA to the datafiles and redo log files is skipped. The subsequent reopening of the database requires crash recovery, which Oracle performs automatically.

Note: If a system crash or power failure occurs while the database is open, the instance is, in effect, aborted, and crash recovery is performed when the database is reopened.

Unmount a Database

Once the database is closed, Oracle unmounts the database to disassociate it from the instance. At this point, the instance remains in the memory of your computer.

After a database is unmountunmounted, Oracle closes the control files of the database.

Shut Down an Instance

The final step in database shutdown is shutting down the instance. When you shut down an instance, the SGA is removed from memory and the background processes are terminated.

Abnormal Instance Shutdown

In unusual circumstances, shutdown of an instance might not occur cleanly; all memory structures might not be removed from memory or one of the background processes might not be terminated. When remnants of a previous instance exist, subsequent instance startup most likely will fail. In such situations, the database administrator can force the new instance to start up by first removing the remnants of the previous instance and then starting a new instance, or by issuing a `SHUTDOWN ABORT` statement in Oracle Enterprise Manager.

> **See Also:** *Oracle9i Database Administrator's Guide* for more detailed information about instance and database startup and shutdown

7

Distributed Processing

This chapter defines distributed processing and describes how the Oracle server and database applications work in a distributed processing environment. This material applies to almost every type of Oracle database system environment.

This chapter includes:

- Client/Server Architecture
- Multitier Architecture
- Distributed Processing Overview
- Oracle Net Services
- Oracle Internet Directory

Client/Server Architecture

In the Oracle database system environment, the database application and the database are separated into two parts: a front-end or **client** portion, and a back-end or **server** portion—hence the term **client/server architecture.** The client executes the database application that accesses database information and interacts with a user through the keyboard, screen, and pointing device such as a mouse. The server executes the Oracle software and handles the functions required for concurrent, shared data access to an Oracle database.

Although the client application and Oracle can be executed on the same computer, greater efficiency can often be achieved when the client portion(s) and server portion are executed by different computers connected through a network. The following sections discuss possible variations in the Oracle client / server architecture.

Multitier Architecture

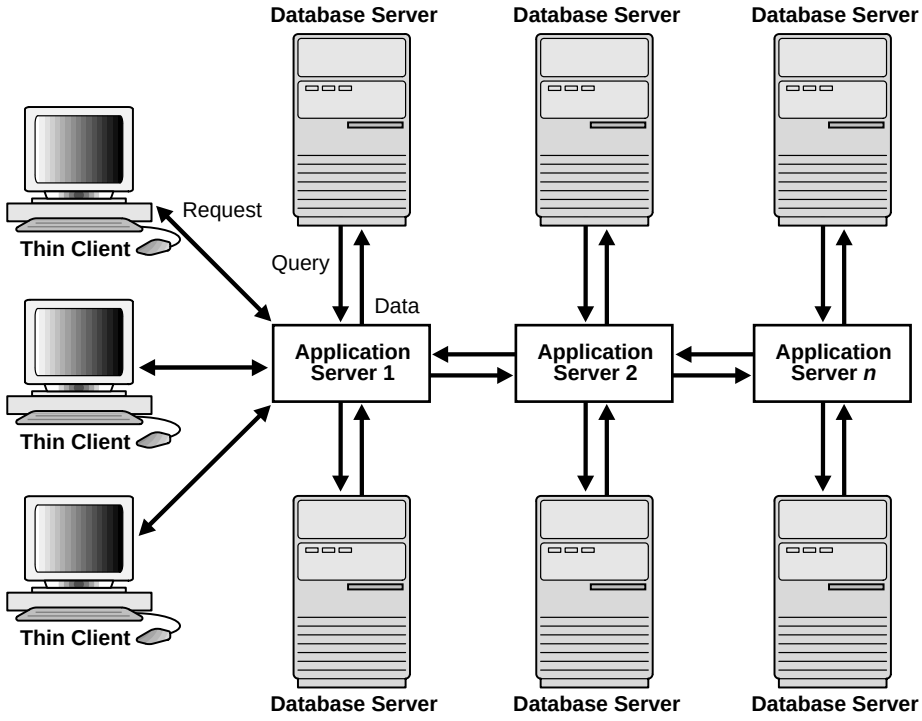
In a multitier architecture environment, an application server provides data for clients and serves as an interface between clients and database servers. This architecture is particularly important because of the prevalence of Internet use.

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 7–1.

Figure 7–1 A Multitier Architecture Environment Example



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 process. 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, which provides an additional level of security. It can also perform some of the query processing for the client, thus removing some of the load from the database server.

The application server assumes the identity of the client when it is performing operations on the database server for that client. The application server's privileges are 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 server does all of the remaining query processing.

The Oracle database server can audit operations performed by the application server on behalf of individual clients as well as operations performed by the application server on its own behalf. For example, a client operation can be a request for information to be displayed on the client, whereas an application server operation can be a request for a connection to the database server.

> **See Also:** "Multitier Authentication and Authorization" on page 24-10 for more information about security issues in a multitier environment

Distributed Processing Overview

Distributed processing is the use of more than one processor to perform the processing for an individual task. Examples of distributed processing in Oracle database systems appear in Figure 7–2.

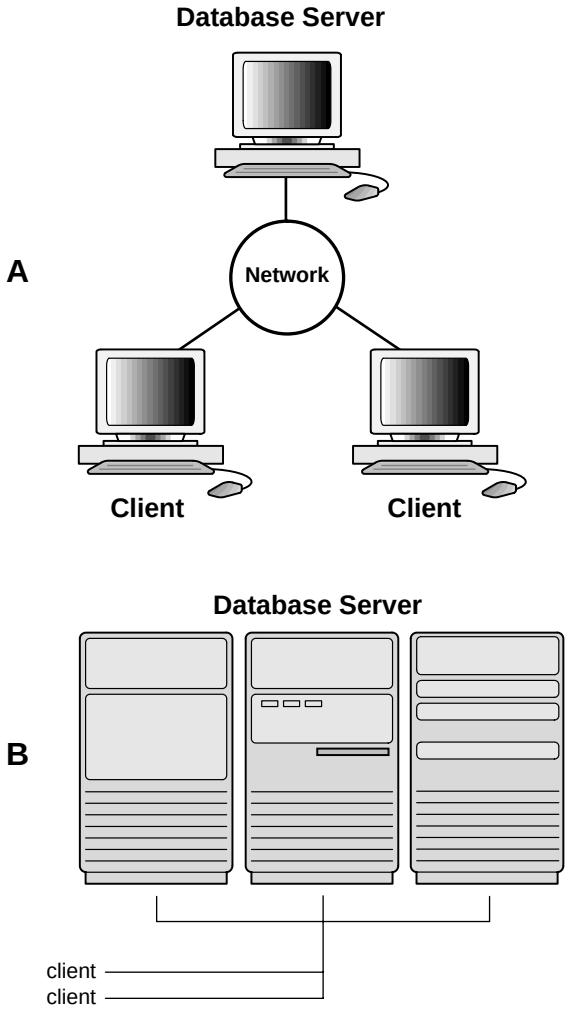
- In Part A of the figure, the client and server are located on different computers, and these computers are connected through a network. The server and clients of an Oracle database system communicate through Oracle Net Services, Oracle's network interface.
- In Part B of the figure, a single computer has more than one processor, and different processors separate the execution of the client application from Oracle.

> **Note:** This chapter applies to environments with one database on one server. In a **distributed database**, one server (Oracle) may need to access a database on another server.

See Also:

- "Oracle Net Services" on page 7-8 for more information about Oracle Net Services
- *Oracle9i Database Administrator's Guide* for more information about clients and servers in distributed databases

Figure 7–2 The Client/Server Architecture and Distributed Processing



Oracle client/server architecture in a distributed processing environment provides the following benefits:

- Client applications are not responsible for performing any 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. If the data is moved or distributed to other database servers, the application continues to function with little or no modification.
- Oracle 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), and 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.
- If necessary, Oracle can be **scaled** *a*s your system grows. You can add multiple servers to distribute the database processing load throughout the network (**horizontally scaled**), or you can move Oracle to a minicomputer or mainframe, to take advantage of a larger system's performance (**vertically scaled**). In either case, all data and applications are maintained with little or no modification, as Oracle is portable between systems.
- In networked environments, shared data is stored on the servers rather than on all computers in the system. This makes it easier and more efficient to manage concurrent access.
- In networked environments, client applications submit database requests to the server using SQL statements. Once received, the SQL statement is processed by the server, and the results are returned to the client application. Network traffic is kept to a minimum because only the requests and the results are shipped over the network.

Oracle Net Services

Oracle Net Services provides enterprise-wide connectivity solutions in distributed, heterogeneous computing environments. Oracle Net Services enables a network session from a client application to an Oracle database.

Oracle Net Services uses the communication protocols or application programmatic interfaces (APIs) supported by a wide range of networks to provide a distributed database and distributed processing for Oracle.

- A communication protocol is a set of rules that determine how applications access the network and how data is subdivided into packets for transmission across the network.
- An API is a set of subroutines that provide, in the case of networks, a means to establish remote process-to-process communication through a communication protocol.

The following sections introduce several Oracle Net Services solutions in a typical network configuration.

Connectivity

Once a network session is established, Oracle Net Services acts as a data courier for the client application and the database server. It is responsible for establishing and maintaining the connection between the client application and database server, as well as exchanging messages between them. Oracle Net Services is able to perform these jobs because it is located on each computer in the network.

Manageability

Oracle Net Services provides location transparency, centralized configuration and management, and quick out-of-the-box installation and configuration.

Internet Scalability

Oracle Net Services enables you to maximize system resources and improve performance. Oracle's **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 for more important tasks.

Internet Security

Network security is enhanced with features like database access control and Oracle Advanced Security.

> **See Also:** *Oracle9i Net Services Administrator's Guide* for more information about these features

How Oracle Net Services Works

Oracle's support of industry network protocols provides an interface between Oracle processes running on the database server and the user processes of Oracle applications running on other computers of the network.

The Oracle protocols take SQL statements from the interface of the Oracle applications and package them for transmission to Oracle through one of the supported industry-standard higher level protocols or programmatic interfaces. The protocols also take replies from Oracle and package them for transmission to the applications through the same higher level communications mechanism. This is all done independently of the network operating system.

Depending on the operation system that executes Oracle, the Oracle Net Services software of the database server could include the driver software and start an additional Oracle background process.

> **See Also:** *Oracle9i Net Services Administrator's Guide* for more information about how Oracle Net Services works

The Listener

When an instance starts, a **listener process** establishes a communication pathway to Oracle. When a user process makes a connection request, the listener determines whether it should use a shared server dispatcher process or a dedicated server process and establishes an appropriate connection.

The listener also establishes a communication pathway between databases. When multiple databases or instances run on one computer, as in Oracle9*i* Real Application Clusters, **service names** enable instances to register automatically with other listeners on the same machine. A service name can identify multiple instances, and an instance can belong to multiple services. Clients connecting to a service do not have to specify which instance they require.

Service Information Registration

Dynamic service registration reduces the administrative overhead for multiple databases or instances. Information about the services to which the listener forwards client requests is registered with the listener. Service information can by dynamically registered with the listener through a feature called **service registration** or statically configured in the `listener.ora` file.

Service registration relies on the **PMON process**—an instance background process—to register instance information with a listener, as well as the current state and load of the instance and **shared server** dispatchers. The registered information enables the listener to forward client connection requests to the appropriate service handler. Service registration does not require configuration in the `listener.ora` file.

The initialization parameter `SERVICE_NAMES` identifies which database 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 usage and current connection counts to all of the listeners in the same services. This enables dynamic load balancing and connection failover.

> **See Also:** *Oracle9i Net Services Administrator's Guide* for more information about the listener

Oracle Internet Directory

Oracle Internet Directory is a directory service implemented as an application on the Oracle9*i* database. It enables retrieval of information about dispersed users and network resources. Oracle Internet Directory combines Lightweight Directory Access Protocol (LDAP), Version 3, the open Internet standard directory access protocol, with the high performance, scalability, robustness, and availability of the Oracle9*i* Server.

Oracle Internet Directory includes the following:

- Oracle directory server, which responds to client requests for information about people and resources, and to updates of that information, using a multitier architecture directly over TCP/IP
- Oracle directory replication server, which replicates LDAP data between Oracle directory servers
- Oracle Directory Manager, a graphical user interface administration tool

- A variety of command line administration and data management tools

See Also:

- "Shared Server Architecture" on page 9-15 for more information about server processes
- "Dedicated Server Configuration" on page 9-21 for more information about server processes
- *Oracle9i Net Services Administrator's Guide* for more information about the listener
- *Oracle9i Real Application Clusters Installation and Configuration* and *Oracle9i Real Application Clusters Deployment and Performance* for information about instance registration and client/service connections in Oracle9*i* Real Application Clusters
- *Oracle Internet Directory Administrator's Guide*

8

Memory Architecture

This chapter discusses the memory architecture of an Oracle instance. It includes:

- Introduction to Oracle Memory Structures
- System Global Area (SGA) Overview
- Program Global Areas (PGA) Overview
- Dedicated and Shared Servers
- Software Code Areas

Introduction to Oracle Memory Structures

Oracle uses memory to store information such as the following:

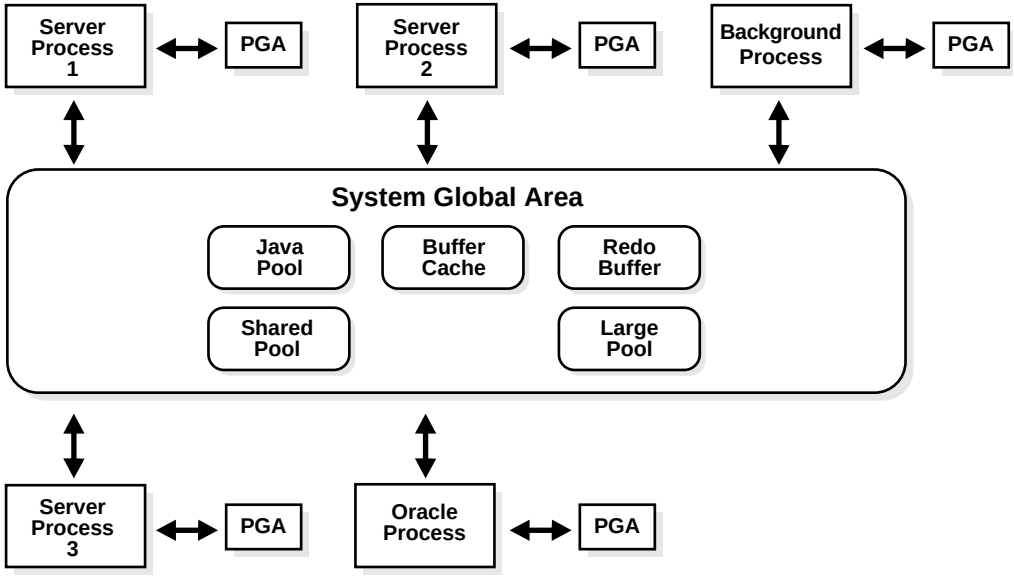
- Program code being executed
- Information about a connected session, even if it is not currently active
- Information needed during program execution (for example, the current state of a query from which rows are being fetched)
- Information that is shared and communicated among Oracle processes (for example, locking information)
- Cached data that is also permanently stored on peripheral memory (for example, data blocks and redo log entries)

The basic memory structures associated with Oracle include:

- System Global Area (SGA), which is shared by all server and background processes and holds the following:
  - Database buffer cache
  - Redo log buffer
  - Shared pool
  - Large pool (if configured)
- Program Global Areas (PGA), which is private to each server and background process; there is one PGA for each process. The PGA holds the following:
  - Stack areas
  - Data areas

Figure 8–1 illustrates the relationships among these memory structures.

Figure 8–1 Oracle Memory Structures



Software Code Areas are another basic memory structure, discussed on page 8-21.

See Also:

- System Global Area (SGA) Overview on page 8-3
- Program Global Areas (PGA) Overview on page 8-16

System Global Area (SGA) Overview

A **system global area** (SGA) is a group of shared memory structures that contain data and control information for one Oracle database instance. If multiple users are concurrently connected to the same instance, then the data in the instance's SGA is shared among the users. Consequently, the SGA is sometimes called the **shared global area**.

An SGA and Oracle processes constitute an Oracle instance. Oracle automatically allocates memory for an SGA when you start an instance, and the operating system reclaims the memory when you shut down the instance. Each instance has its own SGA.

The SGA is read / write. All users connected to a multiple-process database instance can read information contained within the instance's SGA, and several processes write to the SGA during execution of Oracle.

The SGA contains the following data structures:

- Database buffer cache
- Redo log buffer
- Shared pool
- Java pool
- Large pool (optional)
- Data dictionary cache
- Other miscellaneous information

Part of the SGA contains general information about the state of the database and the instance, which the background processes need to access; this is called the **fixed SGA**. No user data is stored here. The SGA also includes information communicated between processes, such as locking information.

If the system uses shared server architecture, then the request and response queues and some contents of the PGA are in the SGA.

See Also:

- "Introduction to an Oracle Instance" on page 6-2 for more information about an Oracle instance
- "Program Global Areas (PGA) Overview" on page 8-16
- "Dispatcher Request and Response Queues" on page 9-17

Dynamic SGA

Beginning with Oracle9*i*, Release 1 (9.0.1), Oracle can change its SGA configuration while the instance is running. With the dynamic SGA infrastructure, the sizes of the buffer cache, the shared pool, and the large pool can be changed without shutting down the instance.

Dynamic SGA also allows Oracle to set, at run time, limits on how much virtual memory Oracle will use for the SGA. Oracle can start instances underconfigured and allow the instance to use more memory by growing the SGA components, up to a maximum of `SGA_MAX_SIZE`. If `SGA_MAX_SIZE` specified in INIT.ORA is less

than the sum of all components specified or defaulted at initialization time, then the setting of SGA_MAX_SIZE in INIT.ORA is ignored.

For optimal performance in most systems, the entire SGA should fit in real memory. If it does not, and virtual memory is used to store parts of it, then overall database system performance can decrease dramatically because portions of the SGA are paged (written to and read from disk) by the operating system. The amount of memory dedicated to all shared areas in the SGA also has performance impact.

The size of the SGA is determined by several initialization parameters. The parameters that most affect SGA size are:

DB_CACHE_SIZE	The size of the cache of standard blocks.
LOG_BUFFER	The number of bytes allocated for the redo log buffer.
SHARED_POOL_SIZE	The size in bytes of the area devoted to shared SQL and PL/SQL statements.
LARGE_POOL_SIZE	The size of the large pool; the default is 0.

The memory allocated for an instance's SGA is displayed on instance startup when using Oracle Enterprise Manager (or SQL*Plus). You can also display the current instance's SGA size by using the SQL*Plus SHOW statement with the SGA clause.

See Also:

- *Oracle9i Database Performance Methods* for discussions of initialization parameters and how they affect the SGA
- *Oracle Enterprise Manager Administrator's Guide* for more information about showing the SGA size with Oracle Enterprise Manager
- *SQL*Plus User's Guide and Reference* for more information about displaying the SGA size with SQL*Plus
- *Oracle9i Database Reference* for information about the V$SGASTAT dynamic view
- Your Oracle installation or user's guide for information specific to your operating system

The Database Buffer Cache

The database buffer cache is the portion of the SGA that holds copies of data blocks read from datafiles. All user processes concurrently connected to the instance share access to the database buffer cache.

The database buffer cache and the shared SQL cache are logically segmented into multiple sets. This organization into multiple sets reduces contention on multiprocessor systems.

Organization of the Database Buffer Cache

The buffers in the cache are organized in two lists: the write list and the least recently used (LRU) list. The **write list** holds dirty buffers, which contain data that has been modified but has not yet been written to disk. The **LRU list** holds free buffers, pinned buffers, and dirty buffers that have not yet been moved to the write list. **Free buffers** do not contain any useful data and are available for use. **Pinned buffers** are currently being accessed.

When an Oracle process accesses a buffer, the process moves the buffer to the most recently used (MRU) end of the LRU list. As more buffers are continually moved to the MRU end of the LRU list, dirty buffers age toward the LRU end of the LRU list.

The first time an Oracle user process requires a particular piece of data, it searches for the data in the database buffer cache. If the process finds the data already in the cache (a **cache hit**), it can read the data directly from memory. If the process cannot find the data in the cache (a **cache miss**), it must copy the data block from a datafile on disk into a buffer in the cache before accessing the data. Accessing data through a cache hit is faster than data access through a cache miss.

Before reading a data block into the cache, the process must first find a free buffer. The process searches the LRU list, starting at the least recently used end of the list. The process searches either until it finds a free buffer or until it has searched the threshold limit of buffers.

If the user process finds a dirty buffer as it searches the LRU list, it moves that buffer to the write list and continues to search. When the process finds a free buffer, it reads the data block from disk into the buffer and moves the buffer to the MRU end of the LRU list.

If an Oracle user process searches the threshold limit of buffers without finding a free buffer, the process stops searching the LRU list and signals the DBW0 background process to write some of the dirty buffers to disk.

> **See Also:** "Database Writer Process (DBWn)" on page 9-8 for more information about DBW*n* processes

The LRU Algorithm and Full Table Scans

When the user process is performing a full table scan, it reads the blocks of the table into buffers and puts them on the LRU end (instead of the MRU end) of the LRU list. This is because a fully scanned table usually is needed only briefly, so the blocks should be moved out quickly to leave more frequently used blocks in the cache.

You can control this default behavior of blocks involved in table scans on a table-by-table basis. To specify that blocks of the table are to be placed at the MRU end of the list during a full table scan, use the `CACHE` clause when creating or altering a table or cluster. You can specify this behavior for small lookup tables or large static historical tables to avoid I/O on subsequent accesses of the table.

> **See Also:** *Oracle9i SQL Reference* for information about the `CACHE` clause

Size of the Database Buffer Cache

Oracle9*i*, Release 1 (9.0.1), supports multiple block size in a database. This is the default block size—the block size used for the system tablespace. You specify the standard block size by setting the initialization parameter `DB_BLOCK_SIZE`. Legitimate values are from 2K to 32K.

To specify the size of the standard block size cache, you set the initialization parameter `DB_CACHE_SIZE`. Optionally, you can also set the size for two additional buffer pools, `KEEP` and `RECYCLE`, by setting `DB_KEEP_CACHE_SIZE` and `DB_RECYCLE_CACHE_SIZE`. These three parameters are independent of one another in Oracle9*i*, Release 1 (9.0.1).

> **See Also:** "Multiple Buffer Pools" on page 8-9 for more information about the `KEEP` and `RECYCLE` buffer pools

The sizes and numbers of non-standard block size buffers are specified by the following parameters:

```
DB_2K_CACHE_SIZE
DB_4K_CACHE_SIZE
DB_8K_CACHE_SIZE
DB_16K_CACHE_SIZE
DB_32K_CACHE_SIZE
```

Each parameter specifies the size of the cache for the corresponding block size.

> **Note:** Platform-specific restrictions regarding the maximum block size apply, so some of these sizes might not be allowed on some platforms.

Example of Setting Block and Cache Sizes

```
db_block_size=4096

db_cache_size=1024M
db_2k_cache_size=256M
db_8k_cache_size=512M
```

In the above example, the parameter `DB_BLOCK_SIZE` sets the standard block size of the database to 4K. The size of the cache of standard block size buffers will be 1024M. Additionally, 2K and 8K caches are also configured, with sizes of 256M and 512M, respectively.

> **Note:** The `DB_`*n*`K_CACHE_SIZE` parameters cannot be used to size the cache for the standard block size. If the value of `DB_BLOCK_SIZE` is *n*K, it is illegal to set `DB_`*n*`K_CACHE_SIZE.` The size of the cache for the standard block size is always determined from the value of `DB_CACHE_SIZE.`

The cache has a limited size, so not all the data on disk can fit in the cache. When the cache is full, subsequent cache misses cause Oracle to write dirty data already in the cache to disk to make room for the new data. (If a buffer is not dirty, it does not need to be written to disk before a new block can be read into the buffer.) Subsequent access to any data that was written to disk results in additional cache misses.

The size of the cache affects the likelihood that a request for data will result in a cache hit. If the cache is large, it is more likely to contain the data that is requested. Increasing the size of a cache increases the percentage of data requests that result in cache hits.

Oracle9*i*, Release 1 (9.0.1), lets you change the size of the buffer cache while the instance is running, without having to shut down the database. You do this with the `ALTER SYSTEM` command. For more information, see "Control of the SGA's Use of Memory" on page 8-15.

You can use the fixed view V$BUFFER_CACHE to track the sizes of the different cache components and any pending resize operations.

The parameter DB_BLOCK_LRU_LATCHES no longer needs to be set in Oracle9*i*, Release 1 (9.0.1). Its value has been obsoleted and is now computed automatically.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for information about tuning the buffer cache

Multiple Buffer Pools

You can configure the database buffer cache with 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. Particular schema objects (tables, clusters, indexes, and partitions) can then be assigned to the appropriate buffer pool to control the way their data blocks age out of the cache.

- The KEEP buffer pool retains the schema object's data blocks in memory.
- The RECYCLE buffer pool eliminates data blocks from memory as soon as they are no longer needed.
- The DEFAULT buffer pool contains data blocks from schema objects that are not assigned to any buffer pool, as well as schema objects that are explicitly assigned to the DEFAULT pool.

The initialization parameters that configure the KEEP and RECYCLE buffer pools are BUFFER_POOL_KEEP and BUFFER_POOL_RECYCLE.

> **Note:** Multiple buffer pools are only available for the standard block size. Non-standard block size caches have a single DEFAULT pool.

> **See Also:**
>
> - *Oracle9i Database Performance Guide and Reference* for more information about multiple buffer pools
> - *Oracle9i SQL Reference* for the syntax of the BUFFER_POOL clause of the STORAGE clause

The Redo Log Buffer

The **redo log buffer** is a circular buffer in the SGA that holds information about changes made to the database. This information is stored in **redo entries**. Redo entries contain the information necessary to reconstruct, or redo, changes made to the database by `INSERT, UPDATE, DELETE, CREATE, ALTER,` or `DROP` operations. Redo entries are used for database recovery, if necessary.

Redo entries are copied by Oracle server processes from the user's memory space to the redo log buffer in the SGA. The redo entries take up continuous, sequential space in the buffer. The background process LGWR writes the redo log buffer to the active online redo log file (or group of files) on disk.

See Also:

- "Log Writer Process (LGWR)" on page 9-9 for more information about how the redo log buffer is written to disk
- *Oracle9i Backup and Recovery Concepts* for information about online redo log files and groups

The initialization parameter `LOG_BUFFER` determines the size (in bytes) of the redo log buffer. In general, larger values reduce log file I/O, particularly if transactions are long or numerous. The default setting is four times the maximum data block size for the host operating system.

The Shared Pool

The shared pool portion of the SGA contains three major areas: library cache, dictionary cache, buffers for parallel execution messages, and control structures.

Note: If the initialization parameter `PARALLEL_AUTOMATIC_TUNING` is set to `TRUE,` these buffers are allocated from the large pool.

The total size of the shared pool is determined by the initialization parameter `SHARED_POOL_SIZE.` The default value of this parameter is 8M on 32-bit platforms and 64M on 64-bit platforms. Increasing the value of this parameter increases the amount of memory reserved for the shared pool.

Library Cache

The library cache includes the shared SQL areas, private SQL areas (in the case of a multiple transaction server), PL/SQL procedures and packages, and control structures such as locks and library cache handles.

Shared SQL areas are accessible to all users, so the library cache is contained in the shared pool within the SGA.

Shared SQL Areas and Private SQL Areas

Oracle represents each SQL statement it executes with a **shared SQL area** and a **private SQL area**. Oracle recognizes when two users are executing the same SQL statement and reuses the shared SQL area for those users. However, each user must have a separate copy of the statement's private SQL area.

Shared SQL Areas A shared SQL area contains the parse tree and execution plan for a given SQL statement. Oracle saves memory by using one shared SQL area for SQL statements executed multiple times, which often happens when many users execute the same application.

Oracle allocates memory from the shared pool when a new SQL statement is parsed, to store in the shared SQL area. The size of this memory depends on the complexity of the statement. If the entire shared pool has already been allocated, Oracle can deallocate items from the pool using a modified LRU (least recently used) algorithm until there is enough free space for the new statement's shared SQL area. If Oracle deallocates a shared SQL area, the associated SQL statement must be reparsed and reassigned to another shared SQL area at its next execution.

See Also:

- "Private SQL Area" on page 8-16
- *Oracle9i Database Performance Guide and Reference*

PL/SQL Program Units and the Shared Pool

Oracle processes PL/SQL program units (procedures, functions, packages, anonymous blocks, and database triggers) much the same way it processes individual SQL statements. Oracle allocates a shared area to hold the parsed, compiled form of a program unit. Oracle allocates a private area to hold values specific to the session that executes the program unit, including local, global, and package variables (also known as package instantiation) and buffers for executing SQL. If more than one user executes the same program unit, then a single, shared

area is used by all users, while each user maintains a separate copy of his or her private SQL area, holding values specific to his or her session.

Individual SQL statements contained within a PL/SQL program unit are processed as described in the previous sections. 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 executes the statement.

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 accesses the data dictionary frequently during SQL statement parsing. This access is essential to the continuing operation of Oracle.

The data dictionary is accessed so often by Oracle that two special locations in memory are designated to hold dictionary data. One area is called the **data dictionary cache**, also known as the **row cache** because it holds data as rows instead of buffers (which hold entire blocks of data). The other area in memory to hold dictionary data is the library cache. All Oracle user processes share these two caches for access to data dictionary information.

See Also:

- Chapter 5, "The Data Dictionary"
- "Library Cache" on page 8-11

Allocation and Reuse of Memory in the Shared Pool

In general, any item (shared SQL area or dictionary row) in the shared pool remains until it is flushed according to a modified LRU algorithm. The memory for items that are not being used regularly is freed if space is required for new items that must be allocated some space in the shared pool. A modified LRU algorithm allows shared pool items that are used by many sessions to remain in memory as long as they are useful, even if the process that originally created the item terminates. As a result, the overhead and processing of SQL statements associated with a multiuser Oracle system is minimized.

When a SQL statement is submitted to Oracle for execution, Oracle automatically performs the following memory allocation steps:

1. Oracle checks the shared pool to see if a shared SQL area already exists for an identical statement. If so, that shared SQL area is used for the execution of the subsequent new instances of the statement. Alternatively, if there is no shared

SQL area for a statement, Oracle allocates a new shared SQL area in the shared pool. In either case, the user's private SQL area is associated with the shared SQL area that contains the statement.

> **Note:** A shared SQL area can be flushed 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 execute its statement, Oracle reparses the statement, and a new shared SQL area is allocated in the shared pool.

2. Oracle allocates a private SQL area on behalf of the session. The location of the private SQL area depends on the type of connection established for the session.

Oracle also flushes a shared SQL area from the shared pool in these circumstances:

- When the `ANALYZE` statement is used to update or delete the statistics of a table, cluster, or index, all shared SQL areas that contain statements referencing the analyzed schema object are flushed from the shared pool. The next time a flushed statement is executed, the statement is parsed 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 that object is later modified in any way, the shared SQL area is **invalidated** (marked invalid), and the statement must be reparsed the next time it is executed.
- If you change a database's global database name, all information is flushed from the shared pool.
- The administrator can manually flush all information in the shared pool to assess the performance (with respect to the shared pool, not the data buffer cache) that can be expected after instance startup without shutting down the current instance. The statement `ALTER SYSTEM FLUSH SHARED_POOL` is used to do this.

See Also:

- "Shared SQL Areas and Private SQL Areas" on page 8-11 for more information about the location of the private SQL area
- Chapter 19, "Dependencies Among Schema Objects" for more information about the invalidation of SQL statements and dependency issues
- *Oracle9i SQL Reference* for information about using `ALTER SYSTEM FLUSH SHARED_POOL`
- *Oracle9i Database Reference* for information about `V$SQL` and `V$SQLAREA` dynamic views

The Large Pool

The database administrator can configure an optional memory area called the **large pool** to provide large memory allocations for:

- Session memory for the shared server and the Oracle XA interface (used where transactions interact with more than one database)
- I/O server processes
- Oracle backup and restore operations
- Parallel execution message buffers, if the initialization parameter `PARALLEL_AUTOMATIC_TUNING` is set to `TRUE` (otherwise, these buffers are allocated to the shared pool)

By allocating session memory from the large pool for shared server, Oracle XA, or parallel query buffers, Oracle can use the shared pool primarily for caching shared SQL and avoid the performance overhead caused by shrinking the shared SQL cache.

In addition, the memory for Oracle backup and restore operations, for I/O server processes, and for parallel buffers is allocated in buffers of a few hundred kilobytes. The large pool is better able to satisfy such large memory requests than the shared pool.

The large pool does not have an LRU list. It is different from reserved space in the shared pool, which uses the same LRU list as other memory allocated from the shared pool.

See Also:

- "Shared Server Architecture" on page 9-15 for information about allocating session memory from the large pool for the shared server
- *Oracle9i Application Developer's Guide - Fundamentals* for information about Oracle XA
- *Oracle9i Database Performance Methods* for more information about the large pool, reserve space in the shared pool, and I/O server processes
- "Degree of Parallelism" on page 20-8 for information about allocating memory for parallel execution

Control of the SGA's Use of Memory

Dynamic SGA provides external controls for increasing and decreasing Oracle's use of physical memory. Together with the dynamic buffer cache, shared pool, and large pool, dynamic SGA allows the following:

- The SGA can grow in response to a DBA command, up to an operating system specified maximum and the `SGA_MAX_SIZE` specification.
- The SGA can shrink in response to a DBA command, to an Oracle prescribed minimum, usually an operating system preferred limit.
- Both the buffer cache and the SGA pools can grow and shrink at runtime according to some internal, Oracle-managed policy.

Other SGA Initialization Parameters

You can use several initialization parameters to control how the SGA uses memory.

Physical Memory

The `LOCK_SGA` parameter locks the SGA into physical memory.

SGA Starting Address

The `SHARED_MEMORY_ADDRESS` and `HI_SHARED_MEMORY_ADDRESS` parameters specify the SGA's starting address at runtime. These parameters are rarely used. They are needed only on platforms that do not specify the SGA's starting address at link time. For 64-bit platforms, `HI_SHARED_MEMORY_ADDRESS` specifies the high order 32 bits of the 64-bit address.

Extended Buffer Cache Mechanism

The USE_INDIRECT_DATA_BUFFERS parameter enables the extended buffer cache mechanism for 32-bit platforms that can support more than 4 GB of physical memory.

However, the dynamic buffer cache feature in Oracle9*i*, Release 1 (9.0.1), requires every buffer to have a valid virtual address. This is because the underlying unit of allocation, a granule, is identified by its virtual address. For this reason, the extended cache feature will be obsolete in future releases of Oracle9*i*.

See Also:

- *Oracle9i Database Reference* for details about the USE_INDIRECT_DATA_BUFFERS parameter
- Your Oracle installation or user's guide for information specific to your operating system

Program Global Areas (PGA) Overview

A **program global area (PGA)** is a memory region which contains data and control information for a server process. It is a nonshared memory created by Oracle when a server process is started. Access to it is exclusive to that server process and is read and written only by Oracle code acting on behalf of it. The total PGA memory allocated by each server process attached to an Oracle instance is also referred to as the **aggregated PGA memory** allocated by the instance.

See Also: "Connections and Sessions" on page 9-4 for information about sessions

Content of the PGA

The content of the PGA memory varies, depending on whether the instance is running the shared server option or not. But generally speaking, the PGA memory can be classified as follows:

Private SQL Area

A private SQL area contains data such as bind information and runtime memory structures. Each session that issues a SQL statement has a private SQL area. Each user that submits the same SQL statement has his or her own private SQL area that uses a single shared SQL area. Thus, many private SQL areas can be associated with the same shared SQL area.

The private SQL area of a cursor is itself divided into two areas whose lifetimes are different:

- The persistent area, which contains, for example, bind information. It will be freed only when the cursor is closed.
- The run-time area, which will be freed when the execution is terminated.

Oracle creates the runtime area as the first step of an execute request. For `INSERT`, `UPDATE`, and `DELETE` statements, Oracle frees the runtime area after the statement has been executed. For queries, Oracle frees the runtime area only after all rows are fetched or the query is canceled.

The location of a private SQL area depends on the type of connection established for a session. If a session is connected through a dedicated server, private SQL areas are located in the server process's PGA. However, if a session is connected through a shared server, part of the private SQL area is kept in the SGA.

See Also:

- "Program Global Areas (PGA) Overview" on page 8-16 for information about the PGA
- "Connections and Sessions" on page 9-4 for more information about sessions
- "SQL Work Areas" on page 8-18 for information about `SELECT` runtimes during a sort, hash-join, bitmap create, or bitmap merge
- *Oracle9i Net Services Administrator's Guide* for an introduction to shared servers

Cursors and SQL Areas The application developer of an Oracle precompiler program or OCI program can explicitly open **cursors**, or handles to specific private SQL areas, and use them as a named resource throughout the execution of the program. Recursive cursors that Oracle issues implicitly for some SQL statements also use shared SQL areas.

The management of private SQL areas is the responsibility of the user process. The allocation and deallocation of private SQL areas depends largely on which application tool you are using, although the number of private SQL areas that a user process can allocate is always limited by the initialization parameter `OPEN_CURSORS`. The default value of this parameter is 50.

A private SQL area continues to exist until the corresponding cursor is closed or the statement handle is freed. Although Oracle frees the runtime area after the statement completes, the persistent area remains waiting. Application developers close all open cursors that will not be used again to free the persistent area and to minimize the amount of memory required for users of the application.

See Also: "Cursors Overview" on page 16-6

Session Memory

Session memory is the memory allocated to hold a session's variables (logon information) and other information related to the session. For a shared server, the session memory is shared and not private.

SQL Work Areas

For complex queries (for example, decision-support queries), a big portion of the runtime area is dedicated to work areas allocated by memory-intensive operators such as the following:

- Sort-based operators (order by, group-by, rollup, window function)
- Hash-join
- Bitmap merge
- Bitmap create

For example, a sort operator uses a work area (sometimes called the sort area) to perform the in-memory sort of a set of rows. Similarly, a hash-join operator uses a work area (also called the hash area) to build a hash table from its left input. If the amount of data to be processed by these two operators does not fit into a work area, the input data is divided into smaller pieces. This allows some data pieces to be processed in memory while the rest are spilled to temporary disk storage to be processed later. Although bitmap operators do not spill to disk when their associated work area is too small, their complexity is inversely proportional to the size of their work area. Thus, these operators will run faster with larger work area.

The size of a work area can be controlled and tuned. Generally, bigger work areas can significantly improve the performance of a particular operator at the cost of higher memory consumption. Optimally, the size of a work area is big enough such to accommodate the input data and auxiliary memory structures allocated by its associated SQL operator. If not, response time will increase, because part of the input data must be spilled to temporary disk storage. In the extreme case, if the size of a work area is far too small compared to the input data size, multiple passes over

the data pieces must be performed. This can dramatically increase the response time of the operator.

PGA Memory Management for Dedicated Mode

In Oracle 8*i* and prior releases, the DBA could control the maximum size of SQL work areas by setting the following parameters: `sort_area_size, hash_area_size, bitmap_merge_area_size` and `create_bitmap_area_size.` Setting these parameters is difficult, because the maximum work area size is ideally selected on the basis of the data input size and the total number of work areas active in the system. These two factors vary a lot from one work area to another and from one time to another. Thus, the various `*_area_size` parameters are hard to tune under the best of circumstances.

Oracle9*i*, Release 1 (9.0.1), introduces a new mode to automatically and globally manage the size of SQL work areas. To enable this mode, the DBA simply needs to specify the total size dedicated to PGA memory for the Oracle instance. This is done be setting the new initialization parameter `pga_aggregate_target.` The specified number (for example, 2G) is a global target for the Oracle instance and Oracle will try to ensure that the total amount of PGA memory allocated across all database server processes never exceeds this target.

When running in this mode, sizing of work areas for all dedicated sessions becomes automatic and all `*_area_size` parameters are ignored for these sessions. At any given time, the total amount of PGA memory available to active work areas on the instance is automatically derived from the parameter `pga_aggregate_target.` Simply speaking, this amount is set to the value of `pga_aggregate_target` minus the PGA memory allocated by other components of the system (for example, PGA memory allocated by sessions). The resulting PGA memory is then allotted to individual active work areas on the basis of their specific memory requirement.

> **Note:** The initialization parameter `WORKAREA_SIZE_POLICY` is a session- and system-level parameter which can take only two values, `MANUAL` or `AUTO.` The default is `AUTO.` The DBA can set `PGA_AGGREGATE_TARGET,` and then switch back and forth from auto to manual memory management mode.

In Oracle9*i*, Release 1 (9.0.1), new `V$` views and columns have been added to provide PGA memory usage statistics. Most of these statistics are enabled when `pga_aggregate_target` is set.

- Statistics on allocation and use of work area memory can be viewed in the following dynamic views:

  V$SYSSTAT
  V$SESSTAT
  V$PGASTAT
  V$SQL_WORKAREA
  V$SQL_WORKAREA_ACTIVE

- Three new columns have been added to the V$PROCESS view to report the PGA memory allocated and used by an Oracle process:

  PGA_USED_MEM
  PGA_ALLOCATED_MEM
  PGA_MAX_MEM

Note: The automatic PGA memory management mode only applies to work areas allocated by dedicated Oracle servers. The size of work areas allocated by shared Oracle servers is still controlled by the old *_area_size parameters, because these work areas are allocated mainly in SGA and not in PGA

See Also:

- *Oracle9i Database Reference* for information about views
- *Oracle9i Database Performance Guide and Reference* for information about using these views

Dedicated and Shared Servers

Memory allocation depends, in some specifics, on whether the system uses dedicated or shared server architecture. Table 8–1 shows the differences.

Table 8–1 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

Table 8–1 Differences in Memory Allocation Between Dedicated and Shared Servers

Memory Area	Dedicated Server	Shared Server
Location of part of the runtime area for SELECT statements	PGA	SGA
Location of the runtime area for DML/DDL statements	PGA	PGA

Software Code Areas

Software code areas are portions of memory used to store code that is being executed or can be executed. Oracle code is stored in a software area that is typically at a different location from users' programs—a more exclusive or protected location.

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. When possible, Oracle code is shared so that all Oracle users can access it without having multiple copies in memory. This results in a saving of real main memory and improves overall performance.

User programs can be shared or nonshared. Some Oracle tools and utilities (such as SQL*Forms and SQL*Plus) can be installed shared, but some cannot. Multiple instances of Oracle can use the same Oracle 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 Oracle operating system specific documentation for more information.

9

Process Architecture

This chapter discusses the processes in an Oracle database system and the different configurations available for an Oracle system. It includes:

- Introduction to Processes
- User Processes Overview
- Oracle Processes Overview
- Shared Server Architecture
- Dedicated Server Configuration
- The Program Interface

Introduction to Processes

All connected Oracle users must execute two modules of code to access an Oracle database instance:

Application or Oracle tool	A database user executes a database application (such as a precompiler program) or an Oracle tool (such as SQL*Plus), which issues SQL statements to an Oracle database.
Oracle server code	Each user has some Oracle server code executing on his or her behalf, which interprets and processes the application's SQL statements.

These code modules are executed by processes. A **process** is a "thread of control" or a mechanism in an operating system that can execute a series of steps. (Some operating systems use the terms **job** or **task**.) A process normally has its own private memory area in which it runs.

Multiple-Process Oracle Systems

Multiple-process Oracle (also called **multiuser Oracle**) uses several processes to execute different parts of the Oracle code and additional processes for the users—either one process for each connected user or one or more processes shared by multiple users. Most database systems are multiuser, because one of the primary benefits of a database is managing data needed by multiple users at the same time.

Each process in an Oracle instance performs a specific job. By dividing the work of Oracle and database applications into several processes, multiple users and applications can connect to a single database instance simultaneously while the system maintains excellent performance.

Types of Processes

The processes in an Oracle system can be categorized into two major groups:

- User processes execute the application or Oracle tool code.
- Oracle processes execute the Oracle server code. They include server processes and background processes.

The process structure varies for different Oracle configurations, depending on the operating system and the choice of Oracle options. The code for connected users can be configured in one of three ways:

Dedicated server (two-task Oracle)	For each user, the database application is run by a different process (a user process) than the one that executes the Oracle server code (a dedicated server process).
Shared server	The database application is run by a different process (a user process) than the one that executes the Oracle server code. Each server process that executes Oracle server code (a **shared server process**) can serve multiple user processes.

Figure 9–1 illustrates a dedicated server configuration. Each connected user has a separate user process, and several background processes execute Oracle.

Figure 9–1 An Oracle Instance

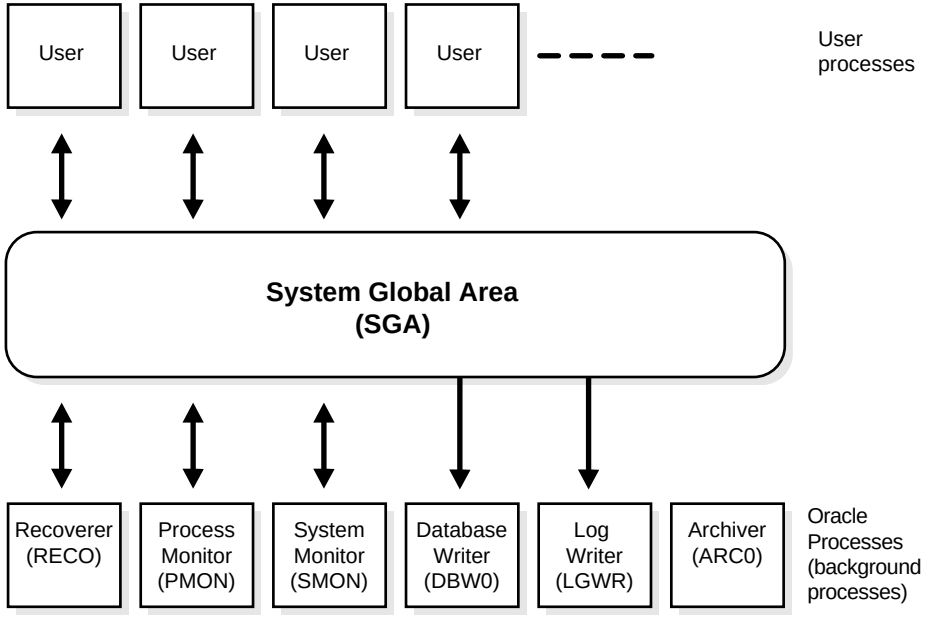


Figure 9–1 can represent multiple concurrent users running an application on the same machine as Oracle. This particular configuration usually runs on a mainframe or minicomputer.

See Also:

- "User Processes Overview" on page 9-4
- "Oracle Processes Overview" on page 9-5
- "Dedicated Server Configuration" on page 9-21
- "Shared Server Architecture" on page 9-15
- Your Oracle operating system specific documentation for more details on configuration choices

User Processes Overview

When a user runs an application program (such as a Pro*C program) or an Oracle tool (such as Oracle Enterprise Manager or SQL*Plus), Oracle creates a **user process** to run the user's application.

Connections and Sessions

Connection and **session** are closely related to **user process** but are very different in meaning.

A **connection** is a communication pathway between a user process and an Oracle instance. A communication pathway is established using available interprocess communication mechanisms (on a computer that executes both the user process and Oracle) or network software (when different computers execute the database application and Oracle, and communicate through a network).

A **session** is a specific connection of a user to an Oracle instance through a user process. For example, when a user starts SQL*Plus, the user must provide a valid username and password, and then a session is established for that user. A session lasts from the time the user connects until the time the user disconnects or exits the database application.

Multiple sessions can be created and exist concurrently for a single Oracle user using the same username. For example, a user with the username/password of `SCOTT/TIGER` can connect to the same Oracle instance several times.

In configurations without the shared server, Oracle creates a server process on behalf of each user session. However, with the shared server, many user sessions can share a single server process.

See Also: "Shared Server Architecture" on page 9-15

Oracle Processes Overview

This section describes the two types of processes that execute the Oracle server code (server processes and background processes). It also describes the trace files and alert file, which record database events for the Oracle processes.

Server Processes

Oracle creates **server processes** to handle the requests of user processes connected to the instance. In some situations when the application and Oracle operate on the same machine, it is possible to combine the user process and corresponding server process into a single process to reduce system overhead. However, when the application and Oracle operate on different machines, a user process always communicates with Oracle through a separate server process.

Server processes (or the server portion of combined user/server processes) created on behalf of each user's application can perform one or more of the following:

- Parse and execute SQL statements issued through the application
- Read necessary data blocks from datafiles on disk into the shared database buffers of the SGA, if the blocks are not already present in the SGA
- Return results in such a way that the application can process the information

Background Processes

To maximize performance and accommodate many users, a multiprocess Oracle system uses some additional Oracle processes called **background processes**.

An Oracle instance can have many background processes; not all are always present. The background processes in an Oracle instance include the following:

- Database Writer (DBW0 or DBW*n*)
- Log Writer (LGWR)
- Checkpoint (CKPT)
- System Monitor (SMON)
- Process Monitor (PMON)
- Archiver (ARC*n*)
- Recoverer (RECO)
- Lock Manager Server (LMS) - Real Application Clusters only

- Queue Monitor (QMN*n*)
- Dispatcher (D*nnn*)
- Server (S*nnn*)

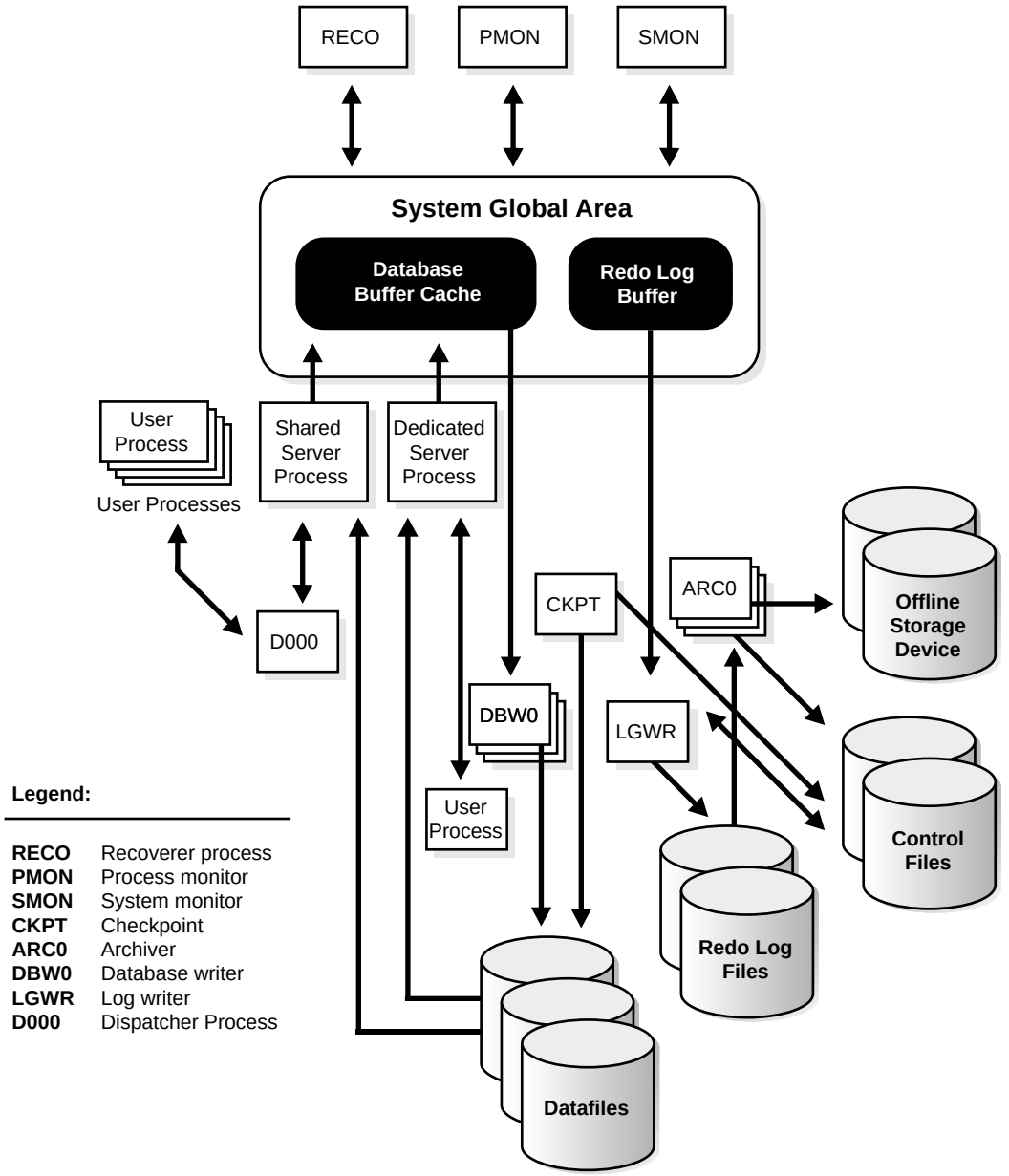
On many operating systems, background processes are created automatically when an instance is started.

Figure 9–2 illustrates how each background process interacts with the different parts of an Oracle database, and the rest of this section describes each process.

See Also:

- *Oracle9i Real Application Clusters Concepts* for more information. Oracle9*i* Real Application Clusters are not illustrated in Figure 9–2
- Your Oracle operating system specific documentation for details on how these processes are created

Figure 9–2 The Background Processes of a Multiple-Process Oracle Instance



Database Writer Process (DBW*n*)

The **database writer process (DBWn)** writes the contents of buffers to datafiles. The DBW*n* processes are responsible for writing modified (dirty) buffers in the database buffer cache to disk. Although one database writer process (DBW0) is adequate for most systems, you can configure additional processes (DBW1 through DBW9) to improve write performance if your system modifies data heavily. These additional DBW*n* processes are not useful on uniprocessor systems.

When a buffer in the database buffer cache is modified, it is marked **dirty**. A **cold** buffer is a buffer that has not been recently used according to the least recently used (LRU) algorithm. The DBW*n* process writes cold, dirty buffers to disk so that user processes are able to find cold, clean buffers that can be used to read new blocks into the cache. As buffers are dirtied by user processes, the number of free buffers diminishes. If the number of free buffers drops too low, user processes that must read blocks from disk into the cache are not able to find free buffers. DBW*n* manages the buffer cache so that user processes can always find free buffers.

By writing cold, dirty buffers to disk, DBW*n* improves the performance of finding free buffers while keeping recently used buffers resident in memory. For example, blocks that are part of frequently accessed small tables or indexes are kept in the cache so that they do not need to be read in again from disk. The LRU algorithm keeps more frequently accessed blocks in the buffer cache so that when a buffer is written to disk, it is unlikely to contain data that will be useful soon.

The initialization parameter `DB_WRITER_PROCESSES` specifies the number of DBW*n* processes. If your system uses multiple DBW*n* processes, adjust the value of the `DB_BLOCK_LRU_LATCHES` parameter so that each DBW*n* process has the same number of latches (LRU buffer lists).The DBW*n* process writes dirty buffers to disk under the following conditions:

- When a server process cannot find a clean reusable buffer after scanning a threshold number of buffers, it signals DBW*n* to write. DBW*n* writes dirty buffers to disk asynchronously while performing other processing.
- DBW*n* periodically writes buffers to advance the **checkpoint**, which is the position in the redo thread (log) from which crash or instance recovery begins. This log position is determined by the oldest dirty buffer in the buffer cache.

In all cases, DBW*n* performs batched (multiblock) writes to improve efficiency. The number of blocks written in a multiblock write varies by operating system.

See Also:

- "The Database Buffer Cache" on page 8-6
- *Oracle9i Database Performance Guide and Reference* for advice on setting `DB_WRITER_PROCESSES` and `DB_BLOCK_LRU_LATCHES` and for information about how to monitor and tune the performance of a single DBW0 process or multiple DBW*n* processes
- *Oracle9i Backup and Recovery Concepts*

Log Writer Process (LGWR)

The **log writer process (LGWR)** is responsible for redo log buffer management—writing the redo log buffer to a redo log file on disk. LGWR writes all redo entries that have been copied into the buffer since the last time it wrote.

The redo log buffer is a circular buffer. When LGWR writes redo entries from the redo log buffer to a redo log file, server processes can then 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 redo log is heavy.

LGWR writes one contiguous portion of the buffer to disk. LGWR writes:

- A commit record when a user process commits a transaction
- Redo log buffers
  - Every three seconds
  - When the redo log buffer is one-third full
  - When a DBW*n* process writes modified buffers to disk, if necessary

---

Note: Before DBW*n* can write a modified buffer, all redo records associated with the changes to the buffer must be written to disk (the **write-ahead protocol**). If DBW*n* finds that some redo records have not been written, it signals LGWR to write the redo records to disk and waits for LGWR to complete writing the redo log buffer before it can write out the data buffers.

LGWR writes synchronously to the active mirrored group of online redo log files. If one of the files in the group is damaged or unavailable, LGWR continues writing to

other files in the group and logs an error in the LGWR trace file and in the system ALERT file. If all files in a group are damaged, or the group is unavailable because it has not been archived, LGWR cannot continue to function.

When a user issues a `COMMIT` statement, LGWR puts a commit record in the redo log buffer and writes it to disk immediately, along with the transaction's redo entries. The corresponding changes to data blocks are deferred until it is more efficient to write them. This is called a **fast commit** mechanism. The atomic write of the redo entry containing the transaction's commit record is the single event that determines the transaction has committed. Oracle returns a success code to the committing transaction, although the data buffers have not yet been written to disk.

Note: Sometimes, if more buffer space is needed, LGWR writes redo log entries before a transaction is committed. These entries become permanent only if the transaction is later committed.

When a user commits a transaction, the transaction is assigned a **system change number (SCN)**, which Oracle records along with the transaction's redo entries in the redo log. SCNs are recorded in the redo log so that recovery operations can be synchronized in Oracle9*i* Real Application Clusters and distributed databases.

In times of high activity, LGWR can write to the online redo log file using **group commits**. For example, assume that a user commits a transaction. LGWR must write the transaction's redo entries to disk, and as this happens, other users issue `COMMIT` statements. However, LGWR cannot write to the online redo log file to commit these transactions until it has completed its previous write operation. After the first transaction's entries are written to the online redo log file, the entire list of redo entries of waiting transactions (not yet committed) can be written to disk in one operation, requiring less I/O than do transaction entries handled individually. Therefore, Oracle minimizes disk I/O and maximizes performance of LGWR. If requests to commit continue at a high rate, then every write (by LGWR) from the redo log buffer can contain multiple commit records.

See Also:

- "The Redo Log Buffer" on page 8-10
- "Trace Files and the Alert Log" on page 9-14
- *Oracle9i Real Application Clusters Deployment and Performance* for more information about SCNs and how they are used
- *Oracle9i Database Administrator's Guide* for more information about SCNs and how they are used
- *Oracle9i Database Performance Guide and Reference f*or information about how to monitor and tune the performance of LGWR

Checkpoint Process (CKPT)

When a checkpoint occurs, Oracle must update the headers of all datafiles to record the details of the checkpoint. This is done by the CKPT process. The CKPT process does not write blocks to disk; DBW*n* always performs that work.

The statistic **DBWR checkpoints** displayed by the System_Statistics monitor in Oracle Enterprise Manager indicates the number of checkpoint requests completed.

See Also: *Oracle9i Real Application Clusters Administration* for information about CKPT with Oracle9*i* Real Application Clusters

System Monitor Process (SMON)

The **system monitor process (SMON)** performs crash recovery, if necessary, at instance startup. SMON is also responsible for cleaning up temporary segments that are no longer in use and for coalescing contiguous free extents within dictionary-managed tablespaces. If any dead transactions were skipped during crash and instance recovery because of file-read or offline errors, SMON recovers them when the tablespace or file is brought back online. SMON wakes up regularly to check whether it is needed. Other processes can call SMON if they detect a need for SMON to wake up.

With Oracle9*i* Real Application Clusters, the SMON process of one instance can perform instance recovery for a failed CPU or instance.

See Also: *Oracle9i Real Application Clusters Administration* for more information about SMON

Process Monitor Process (PMON)

The **process monitor (PMON)** performs process recovery when a user process fails. PMON is responsible for cleaning up the database buffer cache and freeing resources that the user process was using. For example, it resets the status of the active transaction table, releases locks, and removes the process ID from the list of active processes.

PMON periodically checks the status of dispatcher and server processes, and restarts any that have died (but not any that Oracle has terminated intentionally). PMON also registers information about the instance and dispatcher processes with the network listener.

Like SMON, PMON wakes up regularly to check whether it is needed and can be called if another process detects the need for it.

Recoverer Process (RECO)

The **recoverer process (RECO)** is a background process used with the distributed database configuration that automatically resolves failures involving distributed transactions. The RECO process of a node automatically connects to other databases involved in an in-doubt distributed transaction. When the RECO process reestablishes a connection between involved database servers, it automatically resolves all in-doubt transactions, removing from each database's pending transaction table any rows that correspond to the resolved in-doubt transactions.

If the RECO process fails to connect with a remote server, RECO automatically tries to connect again after a timed interval. However, RECO waits an increasing amount of time (growing exponentially) before it attempts another connection.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about distributed transaction recovery

The RECO process is present only if the instance permits distributed transactions and if the `DISTRIBUTED_TRANSACTIONS` parameter is greater than zero. If this initialization parameter is zero, RECO is not created during instance startup.

Job Queue Processes

Job queue processes are used for batch processing. They execute user jobs. They can be viewed as a scheduler service that can be used to schedule jobs as PLSQL statements or procedures on an Oracle instance. Given a start date and an interval, the job queue processes try to execute the job at the next occurrence of the interval.

Beginning with Oracle9*i*, Release 1 (9.0.1), job queue processes are managed dynamically. This allows job queue clients to use more job queue processes when required. The resources used by the new processes are released when they are idle.

Dynamic job queue processes can execute a large number of jobs concurrently at a given interval. The job queue processes execute user jobs as they are assigned by the CJQ process. Here's what happens:

1. The coordinator process, named CJQ0, periodically selects jobs that need to be run from the system `JOB$` table. New jobs selected are ordered by time.
2. The CJQ0 process dynamically spawns job queue slave processes (J000…J999) to execute the jobs.
3. The job queue process executes one of the jobs that was selected by the CJQ process for execution. The processes execute one job at a time.
4. After the process finishes execution of a single job, it polls for more jobs. If there are no scheduled jobs for execution, 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, it aborts after a preset interval.

The init.ora 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 init.ora parameter `JOB_QUEUE_PROCESSES` is set to 0.

See Also: *Oracle9i Database Administrator's Guide* for more information about job queues

Archiver Processes (ARC*n*)

The **archiver process (ARCn)** copies online redo log files to a designated storage device after a log switch has occurred. ARC*n* processes are present only when the database is in ARCHIVELOG mode, and automatic archiving is enabled.

An Oracle instance can have up to 10 ARC*n* processes (ARC0 to ARC9). The LGWR process starts a new ARC*n* process whenever the current number of ARC*n* processes is insufficient to handle the workload. The ALERT file keeps a record of when LGWR starts a new ARC*n* process.

If you anticipate a heavy workload for archiving, such as during bulk loading of data, you can specify multiple archiver processes with the initialization parameter `LOG_ARCHIVE_MAX_PROCESSES`. The `ALTER SYSTEM` statement can change the value of this parameter dynamically to increase or decrease the number of ARC*n* processes. However, you do not need to change this parameter from its default value of 1, because the system determines how many ARC*n* processes are needed, and LGWR automatically starts up more ARC*n* processes when the database workload requires more.

See Also:

- "Trace Files and the Alert Log" on page 9-14
- *Oracle9i Backup and Recovery Concepts*
- Your Oracle operating system specific documentation for details about using the ARC*n* processes

Lock Manager Server Process (LMS)

In Oracle9*i* Real Application Clusters, a Lock Manager Server process (LMS) provides inter-instance resource management.

See Also: *Oracle9i Real Application Clusters Concepts* for more information about this background process

Queue Monitor Processes (QMN*n*)

The **queue monitor process** is an optional background process for Oracle Advanced Queuing, which monitors the message queues. You can configure up to 10 queue monitor processes. These processes, like the Jnnn processes, are different from other Oracle background processes in that process failure does not cause the instance to fail.

See Also: *Oracle9i Application Developer's Guide - Advanced Queuing* for more information about Oracle Advanced Queuing

Trace Files and the Alert Log

Each server and background process can write to an associated **trace file**. When a process detects an internal error, it dumps information about the error to its trace file. If an internal error occurs and information is written to a trace file, the administrator should contact Oracle support.

All filenames of trace files associated with a background process contain the name of the process that generated the trace file. The one exception to this is trace files generated by job queue processes (Jnnn).

Additional information in trace files can provide guidance for tuning applications or an instance. Background processes always write this information to a trace file when appropriate.

Each database also has an **Alert log**. The ALERT file of a database is a chronological log of messages and errors, including the following:

- All internal errors (ORA-600), block corruption errors (ORA-1578), and deadlock errors (ORA-60) that occur
- Administrative operations, such as the SQL statements `CREATE/ALTER/DROP DATABASE/TABLESPACE/ROLLBACK SEGMENT` and the Oracle Enterprise Manager or SQL*Plus statements `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 uses the ALERT file to keep a record of these events as an alternative to displaying the information on an operator's console. (Many systems also display this information on the console.) If an administrative operation is successful, a message is written in the ALERT file as "completed" along with a time stamp.

See Also:

- *Oracle9i Database Performance Guide and Reference* for information about enabling the SQL trace facility
- *Oracle9i Database Error Messages* for information about error messages

Shared Server Architecture

Shared server architecture eliminates the need for a dedicated server process for each connection. A dispatcher directs multiple incoming network session requests to a pool of shared server processes. An idle shared server process from a shared pool of server processes picks up a request from a common queue, which means a small number of shared servers can perform the same amount of processing as many dedicated servers. Also, because the amount of memory required for each

user is relatively small, less memory and process management are required, and more users can be supported.

A number of different processes are needed in a shared server system:

- A network listener process that connects the user processes to dispatchers or dedicated servers (the listener process is part of Oracle Net Services, not Oracle).
- One or more dispatcher processes
- One or more shared server processes

Shared server processes require Oracle Net Services or SQL*Net version 2.

Note: To use shared servers, a user process must connect through Oracle Net Services or SQL*Net version 2, even if the process runs on the same machine as the Oracle instance.

When an instance starts, the network listener process opens and establishes a communication pathway through which users connect to Oracle. Then, 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.

When a user process makes a connection request, the listener examines the request and determines whether the user process can use a shared server process. If so, the listener returns the address of the dispatcher process that has the lightest load, and the user process connects to the dispatcher directly.

Some user processes cannot communicate with the dispatcher, so the network listener process cannot connect them to a dispatcher. In this case, or if the user process requests a dedicated server, the listener creates a dedicated server and establishes an appropriate connection.

See Also:

- "Restricted Operations of the Shared Server" on page 9-20
- *Oracle9i Net Services Administrator's Guide* for more information about the network listener

Internet Scalability

Oracle's shared server architecture increases the scalability of applications and the number of clients simultaneously connected to the database. It can enable existing applications to scale up without making any changes to the application itself.

Dispatcher Request and Response Queues

A request from a user is a single program interface call that is part of the user's SQL statement. When a user makes a call, its dispatcher places the request on the **request queue**, where it is picked up by the next available shared server process.

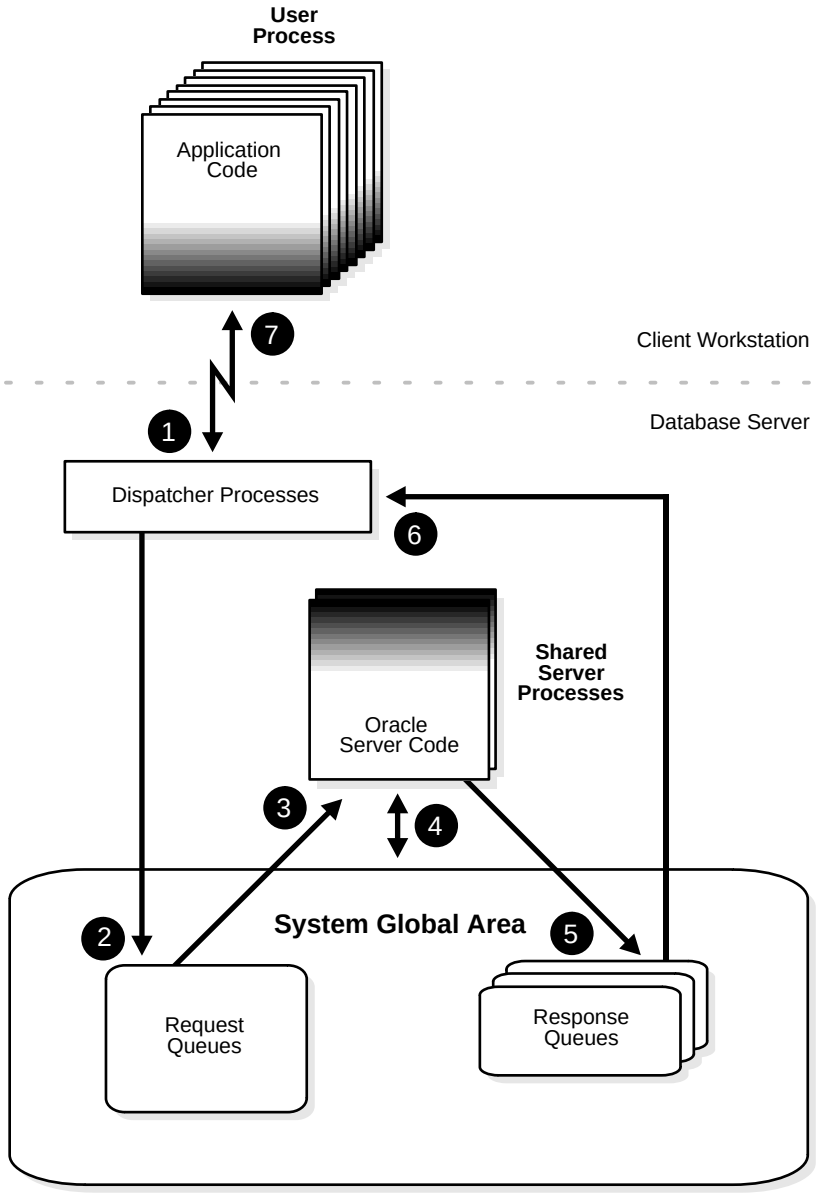
The request queue is in the SGA and is common to all dispatcher processes of an instance. The shared server processes check the common request queue for new requests, picking up new requests on a first-in-first-out basis. One shared server process picks up one request in the queue and makes all necessary calls to the database to complete that request.

When the server completes the request, it places the response on the calling dispatcher's **response queue**. Each dispatcher has its own response queue in the SGA. The dispatcher then returns the completed request to the appropriate user process.

For example, in an order entry system each clerk's user process connects to a dispatcher and each request made by the clerk is sent to that dispatcher, which places the request in the request queue. The next available shared server process picks up the request, services it, and puts the response in the response queue. When a clerk's request is completed, the clerk remains connected to the dispatcher, but the shared server process that processed the request is released and available for other requests. While one clerk is talking to a customer, another clerk can use the same shared server process.

Figure 9–3 illustrates how user processes communicate with the dispatcher across the program interface and how the dispatcher communicates users' requests to shared server processes.

Figure 9–3 The Shared Server Configuration and Shared Server Processes



Dispatcher Processes (D*nnn*)

The **dispatcher processes** support shared server configuration by allowing user processes to share a limited number of server processes. With the shared server, fewer shared server processes are required for the same number of users, Therefore, the shared server can support a greater number of users, particularly in client/server environments where the client application and server operate on different machines.

You can create multiple dispatcher processes for a single database instance. At least one dispatcher must be created for each network protocol used with Oracle. The database administrator starts an optimal number of dispatcher processes depending on the operating system limitation on the number of connections for each process, and can add and remove dispatcher processes while the instance runs.

Note: Each user process that connects to a dispatcher must do so through Oracle Net Services or SQL*Net version 2, even if both processes are running on the same machine.

In a shared server configuration, a network listener process waits for connection requests from client applications and routes each to a dispatcher process. If it cannot connect a client application to a dispatcher, the listener process starts a dedicated server process, and connects the client application to the dedicated server. The listener process is not part of an Oracle instance; rather, it is part of the networking processes that work with Oracle.

See Also:

- "Shared Server Architecture" on page 9-15
- *Oracle9i Net Services Administrator's Guide* for more information about the network listener

Shared Server Processes (S*nnn*)

Each **shared server process** serves multiple client requests in the shared server configuration. Shared server processes and dedicated server processes provide the same functionality, except shared server processes are not associated with a specific user 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 user-related data (which needs to be accessible to all shared server processes). The PGA of a shared server process contains only stack space and process-specific variables.

All session-related information is contained in the SGA. Each shared server process needs to 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. You can limit the amount of space that a session can allocate by setting the resource limit `PRIVATE_SGA` to the desired amount of space in the user's profile.

Oracle dynamically adjusts the number of shared server processes based on the length of the request queue. The number of shared server processes that can be created ranges between the values of the initialization parameters `SHARED_SERVERS` and `MAX_SHARED_SERVERS`.

See Also:

- "Program Global Areas (PGA) Overview" on page 8-16 for more information about the content of a PGA in different types of instance configurations
- Chapter 24, "Controlling Database Access"for more information about resource limits and profiles

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. An error message is issued if you attempt to perform these activities while connected to a dispatcher process.

These activities are typically performed when connected with administrator privileges. When you want to connect with administrator privileges in a system configured with shared servers, you must state in your connect string that you want to use a dedicated server process (`SERVER=DEDICATED`) instead of a dispatcher process.

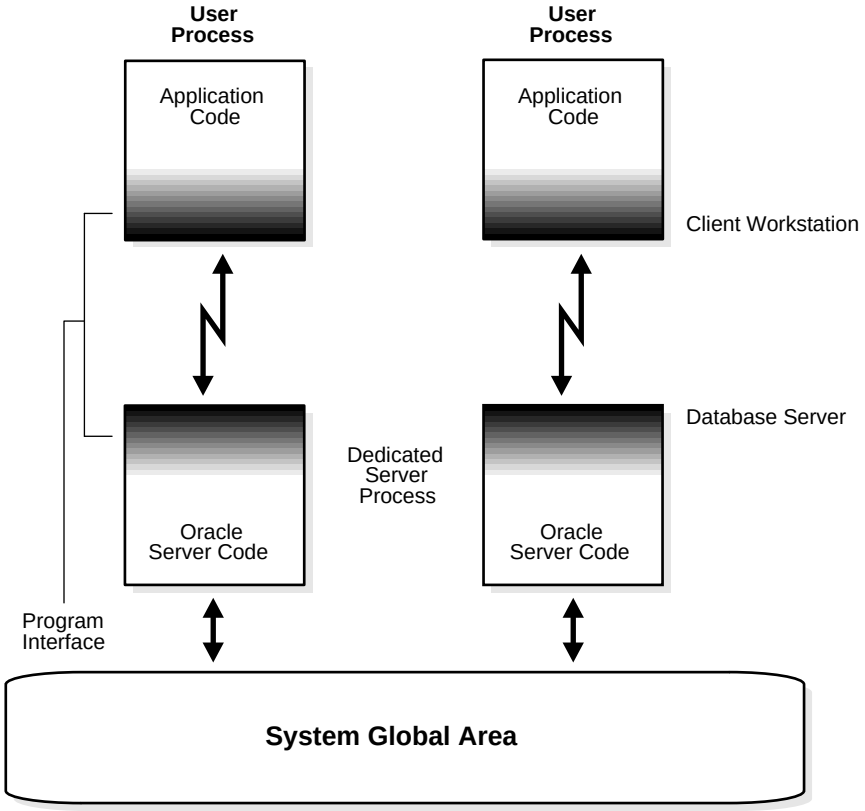
See Also:

- Your Oracle operating system specific documentation
- *Oracle9i Net Services Administrator's Guide* for the proper connect string syntax

Dedicated Server Configuration

Figure 9–4 illustrates Oracle running on two computers using the dedicated server architecture. In this configuration, a user process executes the database application on one machine, and a server process executes the associated Oracle server on another machine.

Figure 9–4 Oracle Using Dedicated Server Processes



The user and server processes are separate, distinct processes. The separate server process created on behalf of each user process is called a **dedicated server process** (or **shadow** process), because this server process acts only on behalf of the associated user process.

This configuration maintains a one-to-one ratio between the number of user processes and server processes. Even when the user is not actively making a database request, the dedicated server process remains (though it is inactive and can be paged out on some operating systems).

Figure 9–4 shows user and server processes running on separate computers connected across a network. However, the dedicated server architecture is also used if the same computer executes both the client application and the Oracle server code but the host operating system could not maintain the separation of the two programs if they were run in a single process. (UNIX is a common example of such an operating system.)

In the dedicated server configuration, the user and server processes communicate using different mechanisms:

- If the system is configured so that the user process and the dedicated server process run on the same computer, the program interface uses the host operating system's interprocess communication mechanism to perform its job.
- If the user process and the dedicated server process run on different computers, the program interface provides the communication mechanisms (such as the network software and Oracle Net Services) between the programs.
- Dedicated server architecture can sometimes result in inefficiency. 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 user process remains idle. The server process is not needed during most of the transaction, and the system is slower for other clerks entering orders. For applications such as this, the shared server architecture may be preferable.

  See Also:

  - Your Oracle operating system specific documentation
  - *Oracle9i Net Services Administrator's Guide*

  for more information about communication links

The Program Interface

The **program interface** is the software layer between a database application and Oracle. The program interface:

- Provides a security barrier, preventing destructive access to the SGA by client user processes

- Acts as a communication mechanism, formatting information requests, passing data, and trapping and returning errors
- Converts and translates data, particularly between different types of computers or to external user program datatypes

The **Oracle code** acts as a server, performing database tasks on behalf of an **application** (a client), such as fetching rows from data blocks. It consists of several parts, provided by both Oracle software and operating system specific software.

Program Interface Structure

The program interface consists of the following pieces:

- Oracle call interface (OCI) or the Oracle runtime library (SQLLIB)
- The client or user side of the program interface (also called the **UPI**)
- Various **Oracle Net Services drivers** (protocol-specific communications software)
- Operating system communications software
- The server or Oracle side of the program interface (also called the *OPI*)

Both the user and Oracle sides of the program interface execute Oracle software, as do the drivers.

Oracle Net Services is the portion of the program interface that allows the client application program and the Oracle server to reside on separate computers in your communication network.

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, and 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 an individual user to use other drivers by specifying the desired driver at the time of connection. Different processes can use different drivers. A single process can have concurrent connections to a single database or to multiple databases (either local or remote) using different Oracle Net Services drivers.

See Also:

- Your system installation and configuration guide for details about choosing, installing, and adding drivers
- Your system Oracle Net Services documentation for information about selecting a driver at runtime while accessing Oracle
- *Oracle9i Net Services Administrator's Guide*

Communications Software for the Operating System

The lowest-level software connecting the user side to the Oracle 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 Corporation but is usually purchased separately from the hardware vendor or a third-party software supplier.

See Also: Your Oracle operating system specific documentation for more information about the communication software of your system

10

Database Resource Management

This chapter describes how Oracle9*i*'s Database Resource Manager works to help a database administrator allocate resources to different groups of users. This chapter includes the following topics:

- Introduction to the Database Resource Manager
- How the Database Resource Manager Works
- Resource Plans and Resource Consumer Groups
- Resource Allocation Methods and Resource Plan Directives
- Interaction with Operating-System Resource Control

Introduction to the Database Resource Manager

Traditionally, it has been up to the operating system to regulate resource management among the various applications running on a system, including Oracle databases. Before Oracle8*i*, there was no way to prioritize one Oracle session over another. The Database Resource Manager gives database administrators more control over resource management decisions, so that resource allocation can be aligned with an enterprise's business objectives.

> **Note:** The Database Resource Manager is available with Oracle Enterprise Edition, beginning with Release 8*i*.

The Database Resource Manager solves many resource allocation problems that an operating system does not manage so well:

- Excessive overhead. This results from operating system context switching between Oracle server processes when the number of server processes is high.
- Inefficient scheduling. The operating system deschedules Oracle database servers while they hold latches, which is inefficient.
- Inappropriate allocation of resources. The operating system distributes resources equally among all active processes and is unable to prioritize one task over another.
- Inability to manage database-specific resources

By using Oracle9*i*'s Database Resource Manager, a database administrator can:

- Guarantee certain users a minimum amount of processing resources regardless of the load on the system and the number of users
- Distribute available processing resources by allocating percentages of CPU time to different users and applications. In a data warehouse, a higher percentage may be given to ROLAP (relational on-line analytical processing) applications than to batch jobs.
- Limit the degree of parallelism of any operation performed by members of a group of users
- Create an **active session pool**. This pool consists of a specified maximum number of user sessions allowed to be concurrently active within a group of users. Additional sessions beyond the maximum are queued for execution, but you can specify a timeout period, after which queued jobs will abort.

- Allow automatic switching of users from one group to another group based on administrator defined criteria. If a member of a particular group of users creates a session that executes for longer than a specified amount of time, that session can be automatically switched to another group of users with different resource requirements.
- Prevent the execution of operations that are estimated to run for a longer time than a predefined limit
- Create an **undo pool**. This pool consists of the amount of undo space that can be consumed in by a group of users.
- Configure an instance to use a particular method of allocating resources. You can dynamically change the method, for example, from a daytime setup to a nighttime setup, without having to shut down and restart the instance.

It is thus possible to balance one user's resource consumption against that of other users and to partition system resources among tasks of varying importance, to achieve overall enterprise goals.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about using the Database Resource Manager

Database Resource Manager Terms

Resources are allocated to users according to a resource plan specified by the database administrator. The following terms are used in specifying a resource plan:

A **resource plan** specifies how the resources are to be distributed among various users (resource consumer groups).

Resource consumer groups allow the administrator to group user sessions together by resource requirements. Resource consumer groups are different from user roles; one database user can have different sessions assigned to different resource consumer groups.

Resource allocation methods determine what policy to use when allocating for any particular resource. Resource allocation methods are used by resource plans and resource consumer groups.

Resource plan directives are a means of assigning consumer groups to particular plans and partitioning resources among consumer groups by specifying parameters for each resource allocation method.

The Database Resource Manager also allows for creation of plans within plans, called subplans. **Subplans** allow further subdivision of resources among different users of an application.

Levels provide a mechanism to specify distribution of unused resources among available users. Up to eight levels of resource allocation can be specified.

Example of a Simple Resource Plan

To illustrate these concepts, take an example of a fictitious company, ABC Inc. ABC sells electronics consumer goods over the Internet. In order ensure the best performance for online customers, at least 85% of the CPU resources should be allocated to them. From the remaining resources, 10% should go to users involved in shipping orders and 5% to billing operations.

To configure an Oracle database to allocate resources in such a way, the DBA creates three resource consumer groups:

- ONLINE for online customers
- SHIPPING for shipping users
- BILLING for billing users

The DBA then creates a resource plan such as the one in Table 10–1.

Table 10–1 Simple Resource Allocation Plan, ABCUSERS

Consumer Group	CPU Resource Allocation
ONLINE	85%
SHIPPING	10%
BILLING	5%

The plan shown in Table 10–1 specifies that 85% of CPU cycles be allotted to ONLINE group sessions, 10% to those of the SHIPPING group and the remaining 5% to the BILLING group. Although this example describes a very simplistic scenario, the Database Resource Manager provides the DBA with a powerful mechanism for implementing controlled resource allocation policies within an Oracle database.

See Also: *PL/SQL User's Guide and Reference* for information about PL/SQL code to create these plans

How the Database Resource Manager Works

The Database Resource Manager controls the distribution of resources among various sessions by controlling the execution schedule inside the database. By controlling which sessions to run and for how long, the Database Resource Manager can ensure that resource distribution matches the plan directive and hence, the business objectives.

Sessions belonging to consumer groups with higher CPU resource allocation are allowed to use more CPU time than sessions belonging to groups or sub plans with lower allocation.

Caution: On UNIX platforms, do not use the `nice` command to alter the operating-system run priorities of processes. Use of this command can lead to instability and unpredictable behavior of the Oracle Server. See "Interaction with Operating-System Resource Control" on page 10-17 for details.

Resource Control

The basic objective of the Database Resource Manager is to maximize system throughput in a way that conforms to business objectives. Consequently, it does not try to enforce CPU allocation percentage limits as long as consumer groups are getting the resources they need.

Example of Resource Control

Consider the plan in Table 10–1. If this plan is activated on a system with a single CPU, any one of the consumer groups can consume up to 100% of CPU resources, providing other groups do not have enough active sessions to consume their allocation. Therefore, if there are no active sessions in the SHIPPING and BILLING groups, the ONLINE group sessions can use 100% of CPU resources, even though their allocation limit is set to 85%.

Similarly, if the database is hosted on system with three CPUs and each group has only one active session, each session runs on one of the three CPUs; in this case, resource allocation is actually 33.33%, no matter how allocation limits are set. However, if all the consumer groups have enough active sessions to consume all

available CPU resources, then the Database Resource Manager enforces the allocation guidelines specified by the plan directive.

Effectiveness of the Database Resource Manager

The effect of the Database Resource Manager is noticeable only in busy environments with high system utilization.

On a multiprocessor systems, processor affinity scheduling at the operating system level can distort CPU allocation on underutilized systems. On a system with multiple CPUs, if one of the CPUs has resources available while others are fully utilized, the operating system attempts to migrate processes from the busy processor's run queue to an underutilized processor. However this does not happen immediately.

On a fully loaded system with enough processes, processor affinity increases performance; this is because invalidating the current CPU cache and loading the new one can be quite expensive. Since most platforms support processor affinity, enough processes must be run to ensure full system utilization.

Database Integration

The Database Resource Manager is fully integrated into the database security system. A PL/SQL package (`DBMS_RESOURCE_MANAGER`) is supplied that allows the DBA to create, update, and delete resource plans and resource consumer groups. The administrator defines a user's default consumer group and what privileges the user has (using the `DBMS_RESOURCE_MANAGER_PRIVS` package). A user or session can switch resource consumer groups (using `DBMS_SESSION.SWITCH_CURRENT_CONSUMER_GROUP`) to change execution priority, if the user has been granted the privilege to switch to that consumer group. In addition, users or sessions can be moved from group to group by the DBA on a production system, dynamically changing the way CPU resources are used.

It is very simple to use the Database Resource Manager in an environment where each application user logs on to the database using a different database username. It is also not very difficult to implement it where applications use generic database login. Since Database Resource Manager actually controls resource utilization at the session level, it is possible to prioritize one session over another, even if both the sessions belong to the same database user. Therefore, it is possible to switch a session to the desired consumer group because of the user's application role, using the `DBMS_SESSION.SWITCH_CURRENT_CONSUMER_GROUP` procedure, as follows:

```
declare default_group varchar2(30);
```

```
begin DBMS_SESSION.SWITCH_CURRENT_CONSUMER_GROUP('desired_consumer_group',
'default_group', FALSE);
end; /
```

Oracle continues to support user resource limits and profiles used with the Database Resource Manager. While the Database Resource Manager balances different requests for service against each other within the defined resource allocation plan, profiles are used to implement hard limits on a user's consumption of resources.

The Database Resource Manager and the Oracle8*i* automatic degree of parallelism (ADOP) feature have been integrated. ADOP attempts to optimize system utilization by automatically adjusting the degree of parallelism for parallel query operations based on current system load and the Database Resource Manager degree of parallelism directive.

Performance Overhead

The Database Resource Manager can effectively manage resources with minimal overhead. In the worst case, the overhead should be no more than 5%. For systems with hundreds of users, Database Resource Manager can actually improve the performance by reducing context switches and latch contention.

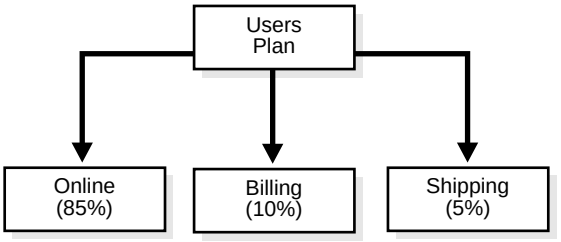
- Database Resource Manager does not switch process as often as any operating system's fair-share scheduler does.
- Database Resource Manager runs fewer processes concurrently and never context-switches a process that is holding a latch.

Judicious use of the Database Resource Manager should not lead to any performance degradation. However, the depth or complexity of a resource plan can impede the process of selecting the process to be executed; therefore, it may advisable to avoid too deep a resource plan. The more levels a plan schema has, the more work the Database Resource Manager must do to pick a session for execution.

Resource Plans and Resource Consumer Groups

A resource plan is a way to group a set of resource consumer groups together and specify how resources should be divided among them. Consider the example from Table 10–1, which can be diagrammed as follows:

Figure 10–1 ABC Resource Allocation Plan, ABCUSERS



Activation of a Resource Plan

You can create as many resource plans as you need in a database. However, only one plan can be active at any given time. You can activate a resource plan in one of two ways: Persistent and Dynamic.

Persistent

Set the value of the `RESOURCE_MANAGER_PLAN` initialization parameter to the plan you want to activate. For example, to activate ABC's simple resource plan, ABCUSERS, you would modify `INIT.ORA` to include the following line:

```
RESOURCE_MANAGER_PLAN='ABCUSERS'
```

When you modify INIT.ORA, you ensure persistence of the resource plan across database shutdown. However, changes in INIT.ORA take effect only when the database is restarted. Use this method to set a the default resource plan for the database.

Dynamic

Issue the `ALTER SYSTEM SET RESOURCE_MANAGER_PLAN` command. Using the same example, you would issue the following command:

```
ALTER SYSTEM SET RESOURCE_MANAGER_PLAN ='ABCUSERS';
```

When you issue the `ALTER SYSTEM SET RESOURCE_MANAGER_PLAN` command, the specified plan is activated immediately, without requiring an instance reboot. However, the database reverts to the default setting in INIT.ORA the next time it is started.

For example, using the dynamic method an administrator could create two different plans, for day time and for night time. The day time plan would allocate more

resources to online users, while the night time plan (when online users are not very active) would ensure higher allocation to batch jobs. Then the DBA uses the `ALTER SYSTEM` command to toggle back and forth between day and night plans without interrupting database services.

The `ALTER SYSTEM SET RESOURCE_MANAGER_PLAN` command is used to dynamically activate, change, or deactivate resource plans.

> **See Also:** Step 3 of "Interaction with Operating-System Resource Control" on page 10-18

Groups of Resource Plans

You can also use resource plans to group other resource plans. This enables you to partition resources among different kinds of applications. For example, the ABC company might need to reserve certain minimum resources to developers and administrators, so that they can perform critical maintenance operations.

Consider a case where at least 25% of the available CPU cycles must be reserved for sessions belonging to two resource consumer groups: DEVELOPERS and ADMINISTRATORS. These CPU cycles should be allocated between DEVELOPERS and ADMINISTRATORS in a ratio of 60 to 40. To achieve this objective, the DBA first creates a maintenance plan with following specifications:

Table 10–2 Sample Maintenance Plan, ABCMAINT

Consumer Group	CPU Resource Allocation
DEVELOPERS	60%
ADMINISTRATORS	40%

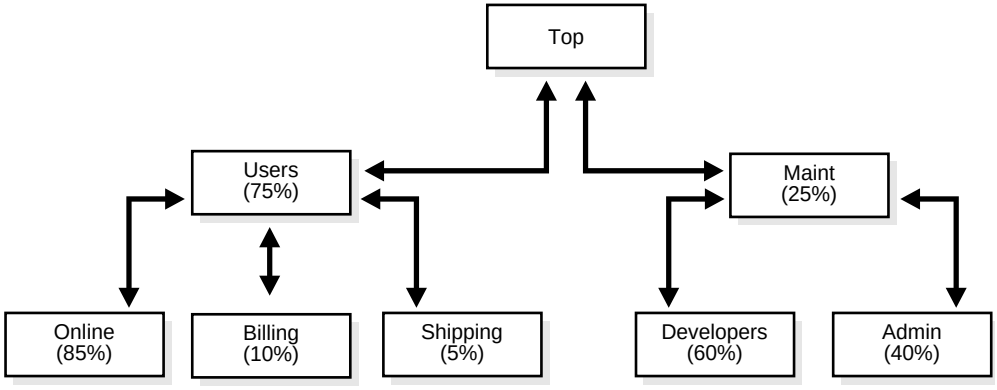
When activated, the plan shown in Table 10–2 ensures that all available resources are distributed among developers' and administrators' sessions in a 60:40 ratio. However, only 25% of all available resources are to be reserved for maintenance; 75% should made available to the ABCUSERS plan. This can be realized by creating a top-level plan with ABCUSERS and ABCMAINT being its members, as shown in Table 10–3:

Table 10–3 Top Plan, ABCTOP

Subplan	CPU Resource Allocation
ABCUSERS	75%
ABCMAINT	25%

The users and maintenance groups become subplans of ABCTOP. The resulting plan tree is diagrammed in Figure 10–2.

Figure 10–2 Resource Plan Containing Subplans



A subplan or consumer group can have more than one parent. For example, in the plan shown in Figure 10–2, the consumer group Admin could very well have been part of both the users and maintenance plans. Because multiple parents are allowed, you can preserve plan independence: You do not need to change anything in subplans when you roll them up to a top-level plan.

A plan tree can have as many hierarchical levels as you want. However, as the number of these levels increase, the overhead associated with resource control increases; the determination of which process is to be run next has to be performed at every level that contains subplans. On the other hand, sub plans provide a good way to partition database resources at a high level among multiple applications and then repartition them within an application among various users. If a given group within an application does not consume its allocation, unused resources are made available to other groups within the same application first. If none of the groups in an application can consume all the resources made available to them, the unused resources are handed back to the parent plan, which can then distribute it among its subplans.

See Also: *PL/SQL User's Guide and Reference* for information about PL/SQL code to create these plans

Resource Allocation Methods and Resource Plan Directives

The Database Resource Manager enables controlled distribution of resources among consumer groups (inter-group), as well within a consumer group (intra-group), using allocation methods and plan directives.

- Plan-level resource allocation methods and directives specify how resources must be distributed among consumer groups.
- Consumer-group methods and directives control resource distribution among sessions belonging to a consumer group.

When scheduling a session for execution, the Database Resource Manager acts as follows:

1. Plan resource allocation guidelines and directives determine which consumer group is to run next.
2. Group-level allocation methods and directives determine which session in the selected group is dispatched to the CPU run queue.

For example, in case of the ABC Company's users plan shown in Table 10–1, plan-level methods and directives specify a resource distribution allowing ONLINE consumer group sessions to be run 85% of the time, while sessions belonging to SHIPPING and BILLING groups get 10% and 5% of CPU time, respectively.

Plan-level directives of ABCUSERS ensure that the ONLINE group is picked up more frequently for execution than the SHIPPING and BILLING groups. However, the ONLINE group usually has several active sessions waiting for execution. Group-level directives determine the order in which these sessions are executed.

Resource Plan Directives

How resources are allocated to resource consumer groups is specified in resource allocation directives. The Database Resource Manager provides several means of allocating resources.

CPU Method

This method lets you specify how CPU resources are to be allocated among consumer groups or subplans. The multiple levels of CPU resource allocation (up to eight levels) provide a means of prioritizing CPU usage within a plan schema. Level

2 gets resources only after level 1 is unable to use all of its resources. Multiple levels not only provide a way of prioritizing, but they provide a way of explicitly specifying how all primary and leftover resources are to be used.

Active Session Pool with Queuing

You can control the maximum number of concurrently active sessions allowed within a consumer group. This maximum designates the active session pool. When a session cannot be initiated because the pool is full, the session is placed into a queue. When an active session completes, the first session in the queue can then be scheduled for execution. You can also specify a timeout period after which a job in the execution queue (waiting for execution) will timeout, causing it to abort with an error.

An entire parallel execution session is counted as one active session.

Degree of Parallelism Limit

*Specifying a parallel degree limit lets you control the maximum degree of parallelism for any operation within a consumer group.

Automatic Consumer Group Switching

This method lets you control resources by specifying criteria that, if met, causes the automatic switching of sessions to another consumer group. The criteria used to determine switching are:

- `SWITCH_GROUP`—specifies the consumer group to which this session is switched if the other (following) criteria are met.
- `SWITCH_TIME`—specifies the length of time that a session can execute before it is switched to another consumer group.
- `SWITCH_ESTIMATE`—specifies whether Oracle is to use its own estimate of how long an operation will execute.

The Database Resource Manager switches a running session to `SWITCH_GROUP` if the session is active for more than `SWITCH_TIME` seconds. Active means that the session is running and consuming resources, not waiting idly for user input or waiting for CPU cycles. The session is allowed to continue running, even if the active session pool for the new group is full. Under these conditions a consumer group can have more sessions running than specified by its active session pool. Once the session finishes its operation and becomes idle, it is switched back to its original group.

If SWITCH_ESTIMATE is set to TRUE, the Database Resource Manager uses a predicted estimate of how long the operation will take to complete. If Oracle's predicted estimate is longer than the value specified as SWITCH_TIME, then Oracle switches the session before execution starts. If this parameter is not set, the operation starts normally and only switches groups when other switch criteria are met.

Execution Time Limit

You can specify a maximum execution time allowed for an operation. If Oracle estimates that an operation will run longer than the specified maximum execution time, the operation is terminated with an error. This error can be trapped and the operation rescheduled.

Undo Pool

You can specify an undo pool for each consumer group. An undo pool controls the amount of total undo that can be generated by a consumer group. When the total undo generated by a consumer group exceeds it's undo limit, the current DML statement generating the redo is terminated. No other members of the consumer group can perform further data manipulation until undo space is freed from the pool.

CPU Resource Allocation

A database administrator can control resource distribution among sessions in competing consumer groups by granting resources at up to eight levels of resource allocation and by specifying how resources are to be distributed among consumer groups at each of these levels.

The users plan shown in Table 10–1 depicts a simple resource distribution scheme using a resource allocation at a single level. This plan can be modified to allocate any unconsumed resources first to administrators (for maintenance operations) and then to any batch jobs. Table 10–4 shows the modified plan:

Table 10–4 Multilevel Users Plan 1

Consumer Group	CPU_LEVEL1	CPU_LEVEL2	CPU_LEVEL3
ONLINE	85%	0%	0%
SHIPPING	10%	0%	0%
BILLING	5%	0%	0%

Table 10–4 Multilevel Users Plan 1

Consumer Group	CPU_LEVEL1	CPU_LEVEL2	CPU_LEVEL3
ADMIN	0%	100%	0%
BATCH	0%	0%	100%

The modified users plan shown in Table 10–4 accomplishes the following:

- CPU_LEVEL1 ensures that at least 85% of CPU resources are available to the sessions belonging to ONLINE, 10% to SHIPPING, and 5% to BILLING consumer groups.
- CPU_LEVEL2 offers to the ADMIN consumer group any resources not consumed by ONLINE, SHIPPING, and BILLING.
- CPU_LEVEL3 makes available to the BATCH consumer group any resources still left.

Multilevel User Plan 1 meets the stated objective by granting resources among the consumer groups at different levels. Sessions belonging to ONLINE, SHIPPING and BILLING groups are always given the first opportunity to run, but their resource consumption is limited to 85%, 10% and 5%, respectively. Any unused resources are made available to Level 2 and are distributed among consumer groups in the proportion of grants made at this level. If there are still any unused resources, they made available to the next level down.

However, the ADMIN group might have to wait a long time, if all the groups at Level 1 are busy. Similarly, the BATCH group might not get to run any sessions at all, if groups at Level 1 and 2 consume all the resources. Such behavior might not be acceptable in some environments. What is required is a plan that allocates most CPU resources to ONLINE, SHIPPING, and BILLING and also ensures availability of certain minimum CPU cycles to the ADMIN and BATCH groups. The modified multilevel users plan is shown in Table 10–5:

Table 10–5 Multilevel Users Plan 2

Consumer Group	CPU_LEVEL1	CPU_LEVEL2	CPU_LEVEL3
ONLINE	75%	0%	0%
SHIPPING	10%	0%	0%
BILLING	5%	0%	0%
ADMIN	0%	80%	0%

Table 10–5 Multilevel Users Plan 2

Consumer Group	CPU_LEVEL1	CPU_LEVEL2	CPU_LEVEL3
BATCH	0%	20%	0%

The modified multiuser plan shown in Table 10–5 accomplishes the following:

- CPU_LEVEL1 now allocates up to 75% of available CPU time to the ONLINE group, while SHIPPING gets 10% and BILLING gets 5%.
- CPU_LEVEL2 splits the remaining 10% of CPU time between ADMIN and BATCH groups in a ratio of 80 to 20. This ensures that ADMIN group sessions will get at least 8% of all available CPU time (80% of 10%); BATCH group sessions will get at least 2%.

Multilevel User Plan 2 guarantees a minimum of 10% of the CPU resources to the ADMIN and BATCH groups. These Level 2 groups will get more CPU time, if the ONLINE, SHIPPING, and BILLING groups do not use all of their allocated resources. If there are no active sessions for any Level 1 groups, the ADMIN group sessions can run 80% of the time and BATCH group sessions can run 20% of the time.

CPU Allocation Rules

The multilevel user plans shown in Table 10–4 and Table 10–5 demonstrate that CPU resource allocation using the Emphasis method follows a set of rules. These rules are as follows:

1. Sessions in resource consumer groups with nonzero percentage allocation at Level 1 always get the first opportunity to run.
2. CPU resources are distributed at a given level by specified percentages.
   - If one consumer group does not consume its allocated resources, unused resources are not given to other groups at the same level, but fall through to the next level.
   - After all resource consumer groups at a given level have had a chance to run, any remaining resources fall through to the next level.
3. The sum of percentages at any given level must be less than or equal to 100.
4. After being offered to consumer groups at all levels, any CPU time that is unused, because of either inactivity or quota restrictions, gets recycled. It is offered to consumer groups again starting at Level 1.

5. Any levels that have no plan directives explicitly specified (for example, Level 3 in the example) are implied to have 0% for all subplans or consumer groups.

The Database Resource Manager allocates CPU resources among groups that have active sessions at a given time. It does not use any historical information in deciding which group to run. For example, say the ONLINE consumer group does not have any active sessions for two hours. During this period, its share of resources is available to other consumer groups. Later, when sessions belonging to the ONLINE consumer group are active, they will still be allocated only 75% of CPU resources. Consumer groups do not accrue credit for the period in which they did not have any active sessions.

Levels and Priorities

Levels are similar to priorities. Consumer groups at Level 1 are offered resources before those at lower levels are considered. Table 10–6 illustrates one way of setting priorities with Database Resource Manager plan directives:

Table 10–6 Simple Priority Plan

Subplan or Group	CPU_LEVEL1	CPU_LEVEL2	CPU_LEVEL3
High Priority	100%	0%	0%
Medium Priority	0%	100%	0%
Low Priority	0%	0%	100%

In Table 10–6, sessions belonging to the Medium Priority group or subplan are allowed to run only when there are no more active sessions in the High Priority group or subplan. Similarly, Low Priority sessions get a chance to run only when there are no active sessions belonging to either the High or Medium priority groups or subplans.

However, a plan like the one shown in Table 10–6 can lead to resource starvation. As long as the High Priority group can use 100% of the CPU resources, no session belonging to the Medium or Low priority groups can run at all. In other words, a set of runaway sessions belonging to the High Priority group could completely stall processing of Medium and Low priority group sessions.

If this is not the effect you intend, you can create a plan that ensures allocation of at least minimum resources to all consumer groups by their relative priorities. For example, you might modify the plan in Table 10–6 as follows:

Table 10–7 Modified Priority Plan

Subplan or Group	CPU_LEVEL1	CPU_LEVEL2	CPU_LEVEL3
High Priority	80%	0%	0%
Medium Priority	10%	0%	0%
Low Priority	10%	0%	0%

The modified plan in Table 10–7 ensures that while the High Priority group gets most of the CPU time, other groups are not completely stalled.

Interaction with Operating-System Resource Control

Oracle9*i* expects a static configuration and allocates internal resources, such as latches, from available resources detected at database startup. The database might not perform optimally and can become unstable if resource configuration changes very frequently. Therefore, operating-system resource control should be used with Oracle databases judiciously, according to the following guidelines:

1. Operating-system resource control should not be used concurrently with the Database Resource Manager, because neither of them are aware of each other's existence. As a result, both the operating system and Database Resource Manager try to control resource allocation in a manner that causes unpredictable behavior and instability of Oracle databases.

   - If you want to control resource distribution within an instance, use the Database Resource Manager and turn off operating-system resource control.
   - If you have multiple instances on a node and you want to distribute resources among them, use operating-system resource control, not the Database Resource Manager.

   Note: Oracle9*i*, Release 1 (9.0.1), does not support the use of both tools simultaneously. Future releases might allow for their interaction on a limited scale.

2. In an Oracle environment, the use of an operating-system resource manager, such as Hewlett Packard's Process Resource Manager (PRM) or Sun's Solaris Resource Manager (SRM), is supported only if all of the following conditions are met:

- Each instance must be assigned to a dedicated operating-system resource manager group or managed entity.
- The dedicated entity running all the instance's processes must run at one priority (or resource consumption) level.
- Process priority management must not be enabled.

Caution: Please note that management of individual Oracle processes at different priority levels (for example, using the `nice` command on UNIX platforms) is not supported. Severe consequences, including instance crashes, can result. You can expect similar undesirable results if operating-system resource control is permitted to manage memory on which an Oracle instance is pinned.

---

3. If you chose to use operating-system resource control, make sure you turn off the Database Resource Manager. By default, the Database Resource Manager is turned off. If it is not, you can turn it off by issuing the following command:

```
ALTER SYSTEM SET RESOURCE_MANAGER_PLAN='';
```

Also remember to reset this parameter in your INIT.ORA file, so that the Database Resource Manager is not activated the next time the database is started up.

Dynamic Reconfiguration

Tools such as Sun's processor sets and dynamic system domains work well with an Oracle database. You might have to restart an instance, if the number of CPUs changes. This is because Oracle discovers the number of CPUs at instance startup, using this information to allocate internal resources. If the number of CPUs changes and performance suffers, rebooting should restore performance.

Part IV

The Object-Relational DBMS

Part IV describes the Oracle relational model for database management and the object extensions to that model.

Part IV contains the following chapters:

- Chapter 11, "Schema Objects"
- Chapter 12, "Partitioned Tables and Indexes"
- Chapter 13, "System-Provided Datatypes"
- Chapter 14, "User-Defined Datatypes"
- Chapter 15, "Object Views"

11

Schema Objects

This chapter discusses the different types of database objects contained in a user's schema. It includes:

- Introduction to Schema Objects
- Tables
- Views
- Materialized Views
- Dimensions
- The Sequence Generator
- Synonyms
- Indexes
- Index-Organized Tables
- Application Domain Indexes
- Clusters
- Hash Clusters

See Also:

- *Oracle9i Database Administrator's Guide*
- "Stored Procedures and Functions" on page 16-24
- Chapter 16, "SQL, PL/SQL, and Java"
- Chapter 18, "Triggers"

for information about additional schema objects

Introduction to Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

- Clusters
- Database links
- Database triggers
- Dimensions
- External procedure libraries
- Index-organized tables
- Indexes
- Indextypes
- Java classes, Java resources, Java sources
- Materialized views
- Materialized view logs
- Object tables
- Object types
- Object views
- Operators
- Packages
- Sequences
- Stored functions, stored procedures
- Synonyms
- Tables
- Views

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

- Contexts

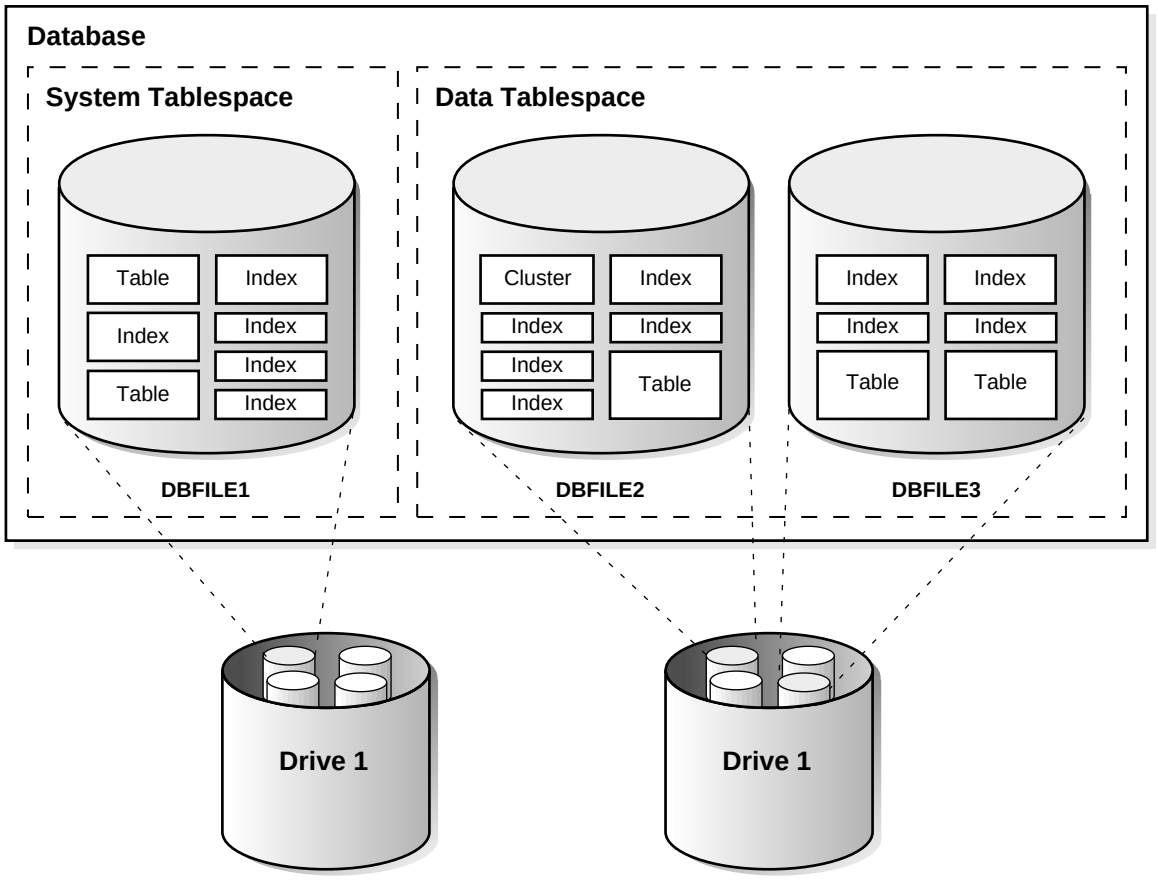
- Directories
- Profiles
- Roles
- Rollback segments
- Tablespaces
- Users

Schema objects are logical data storage structures. Schema objects do not have a one-to-one correspondence to physical files on disk that store their information. However, Oracle stores a schema object logically within a tablespace of the database. The data of each object is physically contained in one or more of the tablespace's datafiles. For some objects, such as tables, indexes, and clusters, you can specify how much disk space Oracle allocates for the object within the tablespace's datafiles.

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.

Figure 11–1 illustrates the relationship among objects, tablespaces, and datafiles.

Figure 11–1 Schema Objects, Tablespaces, and Datafiles



Tables

Tables are the basic unit of data storage in an Oracle database. Data is stored in **rows** and **columns**. You define a table with a **table name** (such as `EMP`) and set of columns. You give each column a **column name** (such as `EMPNO`, `ENAME`, and `JOB`), a **datatype** (such as `VARCHAR2`, `DATE`, or `NUMBER`), and a **width**. The width can be predetermined by the datatype, as in `DATE`. If columns are of the `NUMBER` datatype, define **precision** and **scale** instead of width. A row is a collection of column information corresponding to a single record.

You can 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.

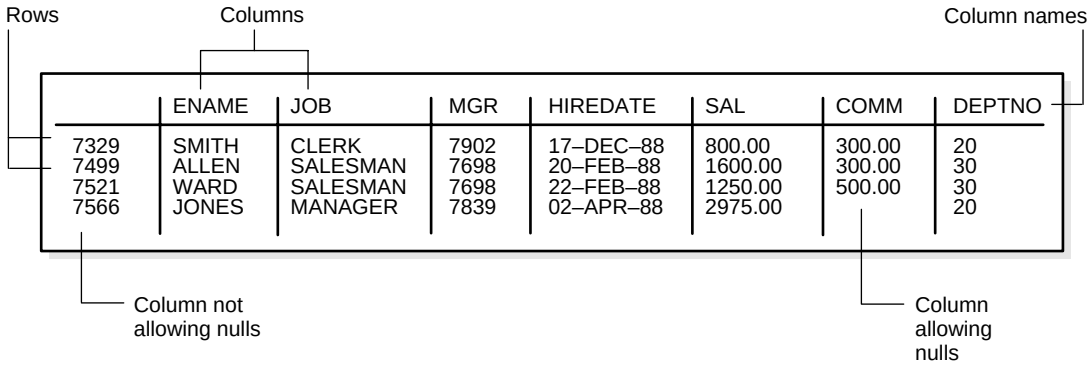
After you create a table, insert rows of data using SQL statements. Table data can then be queried, deleted, or updated using SQL.

Figure 11–2 shows a sample table named `EMP`.

See Also:

- Chapter 13, "System-Provided Datatypes" for a discussion of the Oracle datatypes
- Chapter 23, "Data Integrity" for more information about integrity constraints

Figure 11–2 The EMP Table



How Table Data Is Stored

When you create a table, Oracle automatically allocates a data segment in a tablespace to hold the table's future data. You can control the allocation of space for a table's data segment and use of this reserved space in the following ways:

- You can control the amount of space allocated to the data segment by setting the storage parameters for the data segment.

- You can control the use of the free space in the data blocks that constitute the data segment's extents by setting the PCTFREE and PCTUSED parameters for the data segment.

Oracle stores data for a clustered table in the data segment created for the cluster instead of in a data segment in a tablespace. Storage parameters cannot be specified when a clustered table is created or altered. The storage parameters set for the cluster always control the storage of all tables in the cluster.

The tablespace that contains a nonclustered table's data segment is either the table owner's default tablespace or a tablespace specifically named in the CREATE TABLE statement.

See Also: "User Tablespace Settings and Quotas" on page 24-16

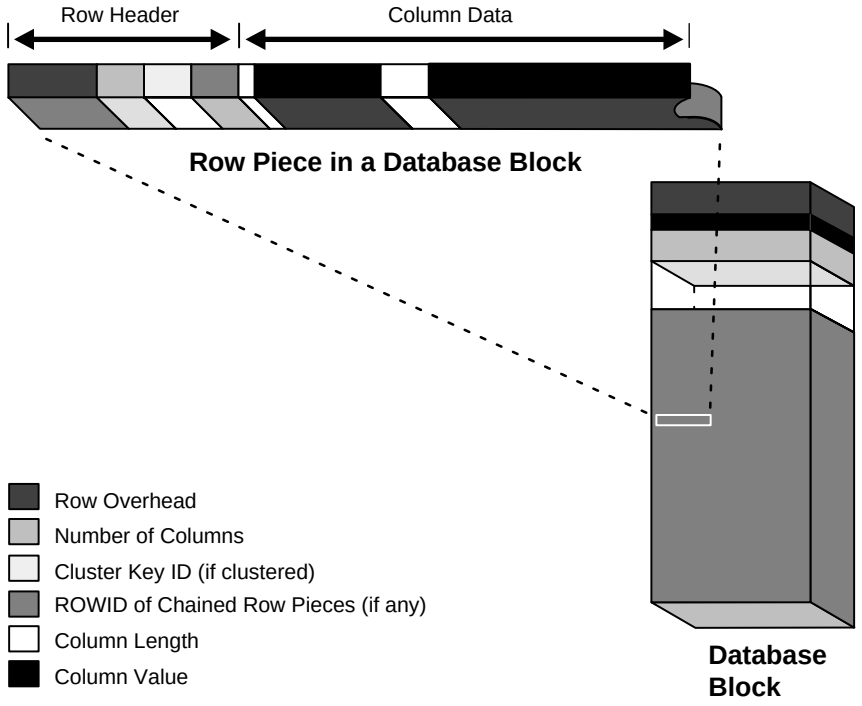
Row Format and Size

Oracle stores each row of a database table containing data for less than 256 columns as one or more row pieces. If an entire row can be inserted into a single data block, then Oracle stores the row as one row piece. However, if all of a row's data cannot be inserted into a single data block or an update to an existing row causes the row to outgrow its data block, Oracle stores the row using multiple row pieces. A data block usually contains only one row piece for each row. When Oracle must store a row in more than one row piece, it is **chained** across multiple blocks.

When a table has more than 255 columns, rows that have data after the 255th column are likely to be chained within the same block. This is called **intra-block chaining**. A chained row's pieces are chained together using the rowids of the pieces. With intra-block chaining, users receive all the data in the same block.If the row fits in the block, users do not see a hit in I/O performance, because there is no extra I/O operation required to retrieve the rest of the row.

Each row piece, chained or unchained, contains a **row header** and data for all or some of the row's columns. Individual columns can also span row pieces and, consequently, data blocks. Figure 11–3 shows the format of a row piece:

Figure 11–3 The Format of a Row Piece



The **row header** precedes the data and contains information about:

- Row pieces
- Chaining (for chained row pieces only)
- Columns in the row piece
- Cluster keys (for clustered data only)

A row fully contained in one block has at least 3 bytes of row header. After the row header information, each row contains column length and data. The column length requires 1 byte for columns that store 250 bytes or less, or 3 bytes for columns that store more than 250 bytes, and precedes the column data. Space required for column data depends on the datatype. If the datatype of a column is variable length, then the space required to hold a value can grow and shrink with updates to the data.

To conserve space, a null in a column only stores the column length (zero). Oracle does not store data for the null column. Also, for trailing null columns, Oracle does not even store the column length.

> **Note:** Each row also uses 2 bytes in the data block header's row directory.

Clustered rows contain the same information as nonclustered rows. In addition, they contain information that references the cluster key to which they belong.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about clustered rows and tables
- "Clusters" on page 11-62
- "Row Chaining and Migrating" on page 3-7
- "Nulls Indicate Absence of Value" on page 11-9
- "Row Directory" on page 3-5

Rowids of Row Pieces

The **rowid** identifies each row piece by its location or address. Once assigned, a given row piece retains its rowid until the corresponding row is deleted or exported and imported using the Export and Import utilities. For clustered tables, if the cluster key values of a row change, then the row keeps the same rowid but also gets an additional pointer rowid for the new values.

Because rowids are constant for the lifetime of a row piece, it is useful to reference rowids in SQL statements such as `SELECT`, `UPDATE`, and `DELETE`.

See Also:

- "Clusters" on page 11-62
- "Physical Rowids" on page 13-18

Column Order

The column order is the same for all rows in a given table. Columns are usually stored in the order in which they were listed in the `CREATE TABLE` statement, but this is not guaranteed. For example, if you create a table with a column of datatype

LONG, then Oracle always stores this column last. Also, if a table is altered so that a new column is added, then the new column becomes the last column stored.

In general, try to place columns that frequently contain nulls last so that rows take less space. Note, though, that if the table you are creating includes a LONG column as well, then the benefits of placing frequently null columns last are lost.

Nulls Indicate Absence of Value

A **null** is the absence of a value in a column of a row. Nulls indicate missing, unknown, or inapplicable data. A null should not be used to imply any other value, such as zero. A column allows nulls unless a NOT NULL or PRIMARY KEY integrity constraint has been defined for the column, in which case no row can be inserted without a value for that column.

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, no information is stored for those columns. In tables with many columns, the columns more likely to contain nulls should be defined last to conserve disk space.

Most comparisons between nulls and other values are by definition neither true nor false, but unknown. To identify nulls in SQL, use the IS NULL predicate. Use the SQL function NVL to convert nulls to non-null values.

Nulls are not indexed, except when the cluster key column value is null or the index is a bitmap index.

See Also:

- *Oracle9i SQL Reference* for more information about comparisons using IS NULL and the NVL function
- "Indexes and Nulls" on page 11-30
- "Bitmap Indexes and Nulls" on page 11-51

Default Values for Columns

You can assign a default value to a column of a table so that when a new row is inserted and a value for the column is omitted or keyword DEFAULT is supplied, a

default value is supplied automatically. Default column values work as though an `INSERT` statement actually specifies the default value.

The datatype of the default literal or expression must match or be convertible to the column datatype.

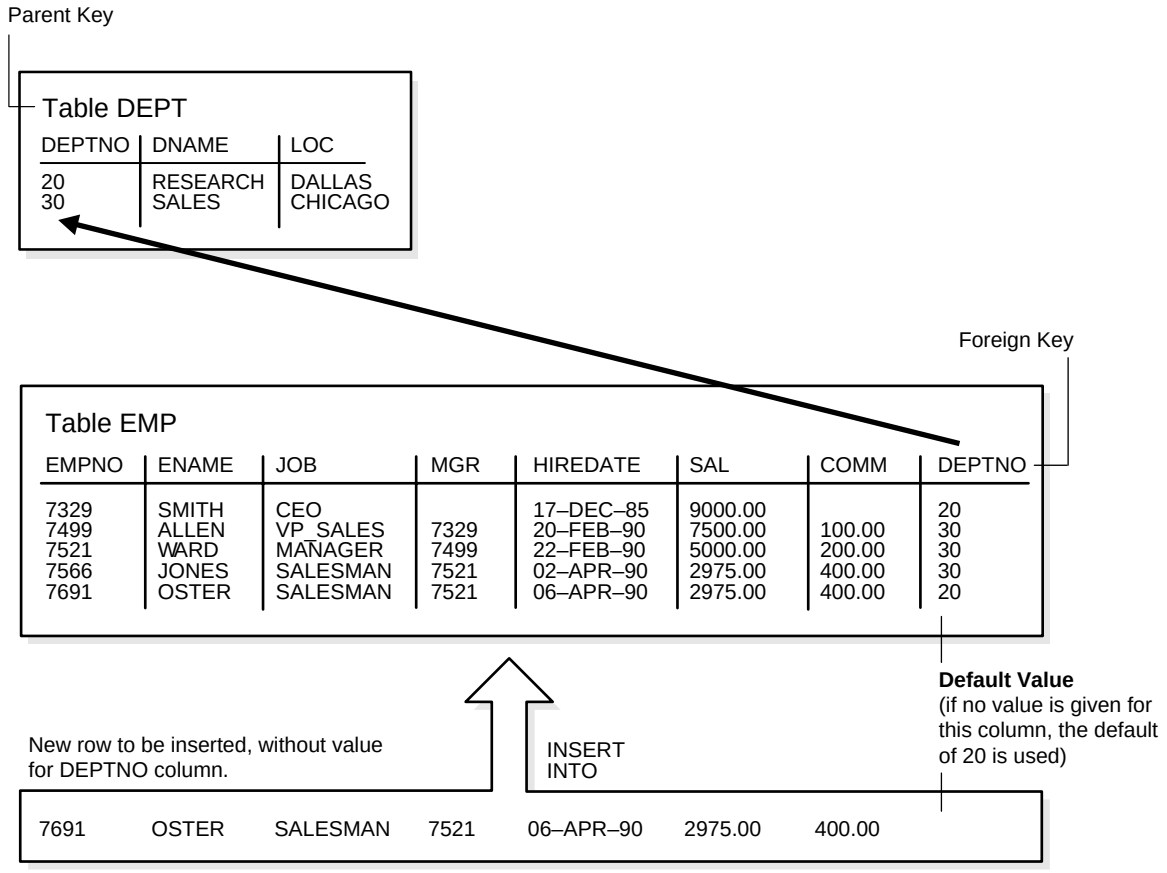
If a default value is not explicitly defined for a column, then the default for the column is implicitly set to `NULL`.

Default Value Insertion and Integrity Constraint Checking

Integrity constraint checking occurs after the row with a default value is inserted. For example, in Figure 11–4, a row is inserted into the `EMP` table that does not include a value for the employee's department number. Because no value is supplied for the department number, Oracle inserts the `DEPTNO` column's default value of 20. After inserting the default value, Oracle checks the `FOREIGN KEY` integrity constraint defined on the `DEPTNO` column.

> **See Also:** Chapter 23, "Data Integrity" for more information about integrity constraints

Figure 11–4 DEFAULT Column Values



Partitioned Tables

Partitioned tables allow your data to be broken down into smaller, more manageable pieces called partitions, or even subpartitions. Indexes can be partitioned in similar fashion. Each partition can be managed individually, and can operate independently of the other partitions, thus providing a structure that can be better tuned for availability and performance.

See Also: Chapter 12, "Partitioned Tables and Indexes"

Nested Tables

You can create a table with a column whose datatype is another table. That is, tables can be **nested** within other tables as values in a column. The Oracle server stores nested table data out of line from the rows of the parent table, using a **store table** that is associated with the nested table column. The parent row contains a unique set identifier value associated with a nested table instance.

See Also:

- "Nested Tables Description" on page 14-12
- *Oracle9i Application Developer's Guide - Fundamentals*

Temporary Tables

In addition to permanent tables, Oracle can create **temporary tables** to hold session-private data that exists only for the duration of a transaction or session.

The `CREATE GLOBAL TEMPORARY TABLE` statement creates a temporary table that can be transaction-specific or session-specific. For transaction-specific temporary tables, data exists for the duration of the transaction. For session-specific temporary tables, data exists for the duration of the session. Data in a temporary table is private to the session. Each session can only see and modify its own data. DML locks are not acquired on the data of the temporary tables. The `LOCK` statement has no effect on a temporary table, because each session has its own private data.

A `TRUNCATE` statement issued on a session-specific temporary table truncates data in its own session. It does not truncate the data of other sessions that are using the same table.

DML statements on temporary tables do not generate redo logs for the data changes. However, undo logs for the data and redo logs for the undo logs are generated. Data from the temporary table is automatically dropped in the case of session termination, either when the user logs off or when the session terminates abnormally such as during a session or instance crash.

You can create indexes for temporary tables using the `CREATE INDEX` statement. Indexes created on temporary tables are also temporary, and the data in the index has the same session or transaction scope as the data in the temporary table.

You can create views that access both temporary and permanent tables. You can also create triggers on temporary tables.

The Export and Import utilities can export and import the definition of a temporary table. However, no data rows are exported even if you use the `ROWS` clause. Similarly, you can replicate the definition of a temporary table, but you cannot replicate its data.

Segment Allocation

Temporary tables use temporary segments. Unlike permanent tables, temporary tables and their indexes do not automatically allocate a segment when they are created. Instead, segments are allocated when the first `INSERT` (or `CREATE TABLE AS SELECT`) is performed. This means that if a `SELECT`, `UPDATE`, or `DELETE` is performed before the first `INSERT`, then the table appears to be empty.

You can perform DDL statements (`ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`, and so on) on a temporary table only when no session is currently bound to it. A session gets bound to a temporary table when an `INSERT` is performed on it. The session gets unbound by a `TRUNCATE`, at session termination, or by doing a `COMMIT` or `ABORT` for a transaction-specific temporary table.

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: "Extents in Temporary Segments" on page 3-18

Parent and Child Transactions

Transaction-specific temporary tables are accessible by user transactions and their child transactions. However, a given transaction-specific temporary table cannot be used concurrently by two transactions in the same session, although it can be used by transactions in different sessions.

If a user transaction does an `INSERT` into the temporary table, then none of its child transactions can use the temporary table afterward.

If a child transaction does an `INSERT` into the temporary table, then at the end of the child transaction, the data associated with the temporary table goes away. After that, either the user transaction or any other child transaction can access the temporary table.

External Tables

Oracle9*i*, Release 1 (9.0.1), provides a way to access data in external sources as if it were in a table in the database. You can connect to the database and create metadata for the external table, using DDL. The DDL for an external table consists of two parts: one part that describes the Oracle column types, another part (the access parameters) which describes the mapping of the external data to the Oracle data columns.

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

External tables are read-only; therefore, no DML operations are possible, and no index can be created on them. You can load Oracle data into flat files and publish it. You can also use external tables to export data in parallel.

The Access Driver

When the database server needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects. Oracle9*i*, Release 1 (9.0.1), provides a default access driver that satisfies most requirements for accessing data in files.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the datatypes for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring the data from the data source is processed so that it matches the definition of the external table.

Data Loading with External Tables

The main use for external tables is to use them as a row source for loading data into a "real" table in the database. After you create an external table, you can then use a `CREATE TABLE AS SELECT` or `INSERT INTO ... AS SELECT` statement, using the external table as the source of the `SELECT` clause.

Note: You cannot insert data into external tables or update records in them; external tables are read-only.

When you access the external table through a SQL statement, the fields of the external table can be used just like any other field in a "normal" table. In particular, you can use the fields as arguments for any SQL built-in function, PL/SQL function, or Java function. This lets you manipulate data from the external source. For data warehousing, you can do more sophisticated transformations in this way than you can with simple datatype conversions. You can also use this mechanism in data warehousing to do data cleansing.

While external tables cannot contain a column object, constructor functions can be used to build a column object from attributes in the external table

Parallel Access to External Tables

Once the metadata for an external table is created, you can query the external data directly and in parallel, using SQL. As a result, the external table acts as a view, which lets you run any SQL query against external data without loading the external data into the database.

The degree of parallel access to an external table is specified using standard parallel hints and with the parallel clause. Using parallelism on an external table allows for concurrent access to the data files that comprise an external table. Whether a single file is accessed concurrently or not is dependent upon the access driver implementation, and attributes of the data file(s) being accessed (for example, record formats or seekable media).

See Also:

- *Oracle9i Database Administrator's Guide* for information about managing external tables, external connections, and directories
- *Oracle9i Database Performance Guide and Reference* for information about tuning loads from external tables
- *Oracle9i Database Utilities* for information about import and export
- *Oracle9i SQL Reference* for information about creating and querying external tables

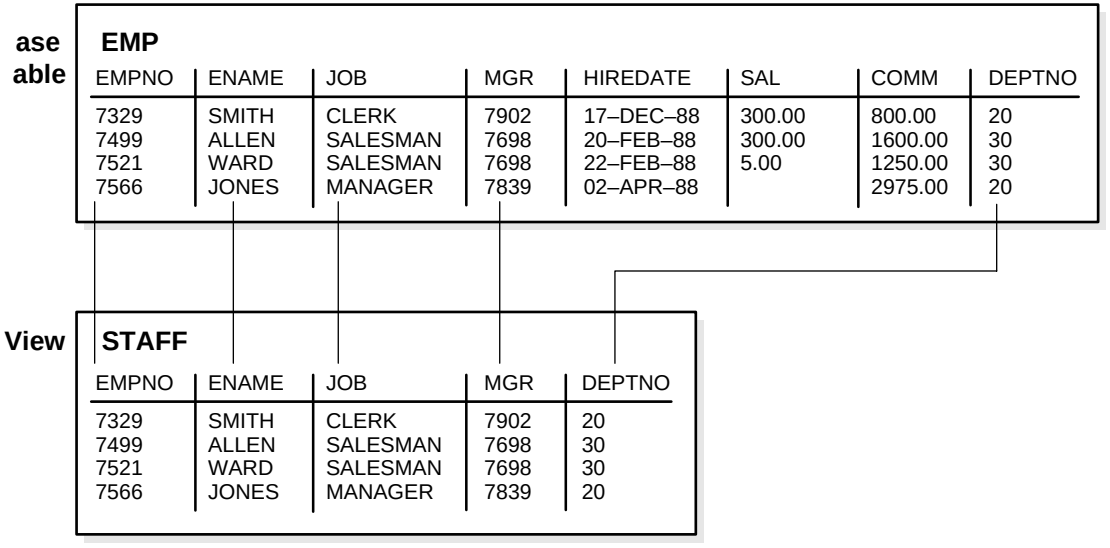
Views

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Therefore, a view can be thought of as a stored query or a virtual table. You can use views in most places where a table can be used.

For example, the EMP table has several columns and numerous rows of information. If you want users to see only five of these columns or only specific rows, then you can create a view of that table for other users to access.

Figure 11–5 shows an example of a view called STAFF derived from the base table EMP. Notice that the view shows only five of the columns in the base table.

Figure 11–5 An Example of a View



Because views are derived from tables, they have many similarities. For example, you can define views with up to 1000 columns, just like a table. You can query views, and with some restrictions you can update, insert into, and delete from views. All operations performed on a view actually affect data in some base table of the view and are subject to the integrity constraints and triggers of the base tables.

Note: You cannot explicitly define triggers on views, but you can define them for the underlying base tables referenced by the view. Oracle Release 1 (9.0.1) does support definition of logical constraints on views.

See Also: *Oracle9i SQL Reference*

Storage for Views

Unlike a table, a view is not allocated any storage space, nor does a view actually contain data. Rather, a view is defined by a query that extracts or derives data from the tables that the view references. These tables are called **base tables**. Base tables can in turn be actual tables or can be views themselves (including materialized views). Because a view is based on other objects, a view requires no storage other than storage for the definition of the view (the stored query) in the data dictionary.

How Views Are Used

Views provide a means to present a different representation of the data that resides within the base tables. Views are very powerful because they let you 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 11–5 shows how the `STAFF` view does not show the `SAL` or `COMM` columns of the base table `EMP`.

- 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.

- Simplify statements for the user

  For example, views allow users to select information from multiple tables without actually knowing how to perform a join.

- 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 a view's defining query references three columns of a four column table, and a fifth column is added to the table, then the view's definition is not affected, and all applications using the view are not affected.

- Express a query that cannot be expressed without using a view

  For example, a view can be defined that joins a `GROUP BY` view with a table, or a view can be defined that joins a `UNION` view with a table.

- Save complex queries

  For example, a query can perform extensive calculations with table information. By saving this query as a view, you can perform the calculations each time the view is queried.

  > **See Also:** *Oracle9i SQL Reference* for information about the `GROUP BY` or `UNION` views

The Mechanics of Views

Oracle stores a view's 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:

1. Merges the statement that references the view with the query that defines the view
2. Parses the merged statement in a shared SQL area
3. Executes the statement

Oracle parses a statement that references a view in a new shared SQL area *only* if no existing shared SQL area contains a similar statement. Therefore, you obtain the benefit of reduced memory usage associated with shared SQL when you use views.

Globalization Support Parameters in Views

When Oracle evaluates views containing string literals or SQL functions that have Globalization Support parameters as arguments (such as `TO_CHAR`, `TO_DATE`, and `TO_NUMBER`), Oracle takes default values for these parameters from the Globalization Support parameters for the session. You can override these default values by specifying Globalization Support parameters explicitly in the view definition.

> **See Also:** *Oracle9i Globalization Support Guide* for information about Globalization Support

Use of Indexes Against Views

Oracle determines whether to use indexes for a query against a view by transforming the original query when merging it with the view's defining query.

Consider the following view:

```
CREATE VIEW emp_view AS
  SELECT empno, ename, sal, loc
```

```
    FROM emp, dept
    WHERE emp.deptno = dept.deptno AND
                       dept.deptno = 10;
```

Now consider the following user-issued query:

```
SELECT ename
  FROM emp_view
  WHERE empno = 9876;
```

The final query constructed by Oracle is:

```
SELECT ename
  FROM emp, dept
  WHERE emp.deptno = dept.deptno AND
        dept.deptno = 10 AND
        emp.empno = 9876;
```

In all possible cases, Oracle merges a query against a view with the view's defining query and those of any underlying views. Oracle optimizes the merged query as if you issued the query without referencing the views. Therefore, Oracle 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.

In some cases, Oracle cannot merge the view definition with the user-issued query. In such cases, Oracle may not use all indexes on referenced columns.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for more information about query optimization

Dependencies and Views

Because a view is defined by a query that references other objects (tables, materialized views, or other views), a view depends on the referenced objects. Oracle automatically handles the dependencies for views. For example, if you drop a base table of a view and then create it again, Oracle determines whether the new base table is acceptable to the existing definition of the view.

> **See Also:** Chapter 19, "Dependencies Among Schema Objects" for a complete discussion of dependencies in a database

Updatable Join Views

A **join view** is defined as a view that has more than one table or view in its FROM clause (a **join**) and that does not use any of these clauses: DISTINCT, AGGREGATION, GROUP BY, START WITH, CONNECT BY, ROWNUM, and set operations (UNION ALL, INTERSECT, and so on).

An **updatable join view** is a join view that involves two or more base tables or views, where UPDATE, INSERT, and DELETE operations are permitted. The data dictionary views ALL_UPDATABLE_COLUMNS, DBA_UPDATABLE_COLUMNS, and USER_UPDATABLE_COLUMNS contain information that indicates which of the view columns are updatable. In order to be inherently updatable, a view cannot contain any of the following constructs:

- A set operator
- A DISTINCT operator
- An aggregate or analytic function
- A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
- A collection expression in a SELECT list
- A subquery in a SELECT list
- Joins (with some exceptions). See *Oracle9i Database Administrator's Guide* for details.

Views that are not updatable can be modified using INSTEAD OF triggers.

See Also:

- *Oracle9i SQL Reference* for further information about updatable views
- "INSTEAD OF Triggers" on page 18-13

Object Views in an Oracle Database

In the Oracle object-relational database, **object views** let you retrieve, update, insert, and delete relational data as if they were stored as object types. You can also define views with columns that are object datatypes, such as objects, REFs, and collections (nested tables and VARRAYs).

See Also:

- Chapter 15, "Object Views"
- *Oracle9i Application Developer's Guide - Fundamentals*

Inline Views

An **inline view** is not a schema object. It is a subquery with an alias (correlation name) that you can use like a view within a SQL statement.

For example, this query joins the summary table `SUMTAB` to an inline view `V` defined on the `TIME` table to obtain `T.YEAR`, and then rolls up the aggregates in `SUMTAB` to the `YEAR` level:

```
SELECT v.year, s.prod_name, SUM(s.sum_sales)
    FROM sumtab s,
        (SELECT DISTINCT t.month, t.year FROM time t) v
    WHERE s.month = v.month
    GROUP BY v.year, s.prod_name;
```

See Also: *Oracle9i SQL Reference* for information about subqueries

Materialized Views

Materialized views are schema objects that can be used to summarize, precompute, replicate, and distribute data. They are suitable in various computing environments such as data warehousing, decision support, and distributed or mobile computing:

- In data warehouses, materialized views are used to precompute and store aggregated data such as sums and averages. Materialized views in these environments are typically referred to as **summaries** because they store summarized data. They can also be used to precompute joins with or without aggregations. If compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher, materialized views can be used for queries that include filter selections.

  Cost-based optimization can use materialized views to improve query performance by automatically recognizing when a materialized view can and should be used to satisfy a request. The optimizer transparently rewrites the request to use the materialized view. Queries are then directed to the materialized view and not to the underlying detail tables or views.

- In distributed environments, materialized views are used to replicate data at distributed sites and synchronize updates done at several sites with conflict

resolution methods. The materialized views as replicas provide local access to data that otherwise has to be accessed from remote sites.

- In mobile computing environments, materialized views are used to download a subset of data from central servers to mobile clients, with periodic refreshes from the central servers and propagation of updates by clients back to the central servers.

Materialized views are similar to indexes in several ways:

- They consume storage space.
- They must be refreshed when the data in their master tables changes.
- They improve the performance of SQL execution when they are used for query rewrites.
- Their existence is transparent to SQL applications and users.

Unlike indexes, materialized views can be accessed directly using a `SELECT` statement. Depending on the types of refresh that are required, they can also be accessed directly in an `INSERT, UPDATE,` or `DELETE` statement.

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:

- "Indexes" on page 11-27
- Chapter 12, "Partitioned Tables and Indexes"
- *Oracle9i Data Warehousing Guide* for information about materialized views in a data warehousing environment

Define Constraints on Views

Data warehousing applications recognize multidimensional data in the Oracle database by identifying Referential Integrity (RI) constraints in the relational schema. RI constraints represent primary and foreign key relationships among tables. By querying the Oracle data dictionary, applications can recognize RI constraints and therefore recognize the multidimensional data in the database. In some environments DBAs, for schema complexity or security reasons, define views on fact and dimension tables. Oracle9*i*, Release 1 (9.0.1), provides the ability to constrain views. By allowing constraint definitions between views, DBAs can propagate base table constraints to the views, thereby allowing applications to recognize multidimensional data even in a restricted environment.

Only logical constraints can be defined on views (that is, declarative and not enforced by Oracle), because the purpose is not to enforce any business rules but to identify multidimensional data. The following constraints can be defined on views:

- Primary key constraint
- Unique constraint
- Referential Integrity constraint

Given that view constraints are declarative, `DISABLE, NOVALIDATE` is the only valid state for a view constraint. However, the `RELY` or `NORELY` state is also allowed, because constraints on views may be used to enable more sophisticated query rewrites; a view constraint in the `RELY` state allows query rewrites to occur when the rewrite integrity level is set to trusted mode.

Note: Although view constraint definitions are declarative in nature, operations on views are subject to the integrity constraints defined on the underlying base tables, and constraints on views can be enforced through constraints on base tables.

Refresh Materialized Views

Oracle maintains the data in materialized views by refreshing them after changes are made to their master tables. The refresh method can be incremental *(***fast refresh***)* or complete. 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.

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.

Materialized View Logs

A **materialized view log** is a schema object that records changes to a master table's 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:

- *Oracle9i Data Warehousing Guide* for information about materialized views and materialized view logs in a warehousing environment
- *Oracle9i Replication* for information about materialized views used for replication

Dimensions

A dimension is a schema object that defines hierarchical relationships between pairs of columns or column sets. A hierarchical relationship is a **functional dependency** from one level of a hierarchy to the next level in the hierarchy. A dimension is a container of logical relationships between columns and does not have any data storage assigned to it.

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 columns in a dimension can come either from the same table *(***denormalized***)* or from multiple tables *(***fully** or **partially normalized***)*. To define a dimension over columns from multiple tables, connect the tables using the `JOIN` clause of the `HIERARCHY` clause.

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 all in the same table. Whether normalized or denormalized, the hierarchical relationships among the columns need to be specified in the `CREATE DIMENSION` statement.

See Also:

- *Oracle9i Data Warehousing Guide* for information about how dimensions are used in a warehousing environment
- *Oracle9i SQL Reference* for information about creating dimensions

The Sequence Generator

The sequence generator provides a sequential series of numbers. The sequence generator is especially useful in multiuser environments for generating unique sequential numbers without the overhead of disk I/O or transaction locking. Therefore, the sequence generator reduces serialization where the statements of two transactions must generate sequential numbers at the same time. By avoiding the serialization that results when multiple users wait for each other to generate and use a sequence number, the sequence generator improves transaction throughput, and a user's wait is considerably shorter.

Sequence numbers are Oracle integers defined in the database of up to 38 digits. A sequence definition indicates general information:

- The name of the sequence
- Whether the sequence ascends or descends
- The interval between numbers
- Whether Oracle should cache sets of generated sequence numbers in memory

Oracle stores the definitions of all sequences for a particular database as rows in a single data dictionary table in the `SYSTEM` tablespace. Therefore, all sequence definitions are always available, because the `SYSTEM` tablespace is always online.

Sequence numbers are used by SQL statements that reference the sequence. You can issue a statement to generate a new sequence number or use the current sequence number. After a statement in a user's session generates a sequence number, the particular sequence number is available only to that session. Each user that references a sequence has access to its own current sequence number.

Sequence numbers are generated independently of tables. Therefore, the same sequence generator can be used for more than one table. Sequence number generation is useful to generate unique primary keys for your data automatically and to coordinate keys across multiple rows or tables. Individual sequence numbers can be skipped if they were generated and used in a transaction that was ultimately

rolled back. Applications can make provisions to catch and reuse these sequence numbers, if desired.

Caution: If accountability for all sequence numbers is required - that is, if your application can never lose sequence numbers - you cannot use Oracle sequences, and you may choose to store sequence numbers in database tables.

Care should be taken when implementing sequence generators using database tables. Even in a single instance configuration, for a high rate of sequence values generation, there will be a performance overhead associated with the cost of locking the row that stores the sequence value.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for performance implications when using sequences
- *Oracle9i SQL Reference* for information about the `CREATE SEQUENCE` statement

Synonyms

A **synonym** is an alias for any table, view, materialized view, sequence, procedure, function, or package. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms are often used for security and convenience. For example, they can do the following:

- Mask the name and owner of an object
- Provide location transparency for remote objects of a distributed database
- Simplify SQL statements for database users

You can create both public and private synonyms. A **public** synonym is owned by the special user group named `PUBLIC` and every user in a database can access it. A **private** synonym is in the schema of a specific user who has control over its availability to others.

Synonyms are very useful in both distributed and nondistributed database environments because they hide the identity of the underlying object, including its location in a distributed system. This is advantageous because if the underlying

object must be renamed or moved, then only the synonym needs to be redefined. Applications based on the synonym continue to function without modification.

Synonyms can also simplify SQL statements for users in a distributed database system. The following example shows how and why public synonyms are often created by a database administrator to hide the identity of a base table and reduce the complexity of SQL statements. Assume the following:

- A table called SALES_DATA is in the schema owned by the user JWARD.
- The SELECT privilege for the SALES_DATA table is granted to PUBLIC.

At this point, you have to query the table SALES_DATA with a SQL statement similar to the following:

```
SELECT * FROM jward.sales_data;
```

Notice how you must include both the schema that contains the table along with the table name to perform the query.

Assume that the database administrator creates a public synonym with the following SQL statement:

```
CREATE PUBLIC SYNONYM sales FOR jward.sales_data;
```

After the public synonym is created, you can query the table SALES_DATA with a simple SQL statement:

```
SELECT * FROM sales;
```

Notice that the public synonym SALES hides the name of the table SALES_DATA and the name of the schema that contains the table.

Indexes

Indexes are optional structures associated with tables and clusters. You can create indexes on one or more columns of a table to speed SQL statement execution on that table. Just as the index in this manual helps you locate information faster than if there were no index, an Oracle index provides a faster access path to table data. Indexes are the primary means of reducing disk I/O when properly used.

You can create many indexes for a table as long as the combination of columns differs for each index. You can create more than one index using the same columns if you specify distinctly different combinations of the columns. For example, the following statements specify valid combinations:

```
CREATE INDEX emp_idx1 ON emp (ename, job);
CREATE INDEX emp_idx2 ON emp (job, ename);
```

You cannot create an index that references only one column in a table if another such index already exists.

Oracle provides several indexing schemes, which provide complementary performance functionality:

- B-tree indexes
- B-tree cluster indexes
- Hash cluster indexes
- Reverse key indexes
- Bitmap indexes
- Bitmap Join Indexes

Oracle also provides support for function-based indexes and domain indexes specific to an application or cartridge.

The absence or presence of an index does not require a change in the wording of any SQL statement. An index is merely a fast access path to the 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.

Indexes are logically and physically independent of the data in the associated table. You can create or drop an index at any time without affecting the base tables or other indexes. If you drop an index, all applications continue to work. However, access of previously indexed data can be slower. Indexes, as independent structures, require storage space.

Oracle automatically maintains and uses indexes after they are created. Oracle automatically reflects changes to data, such as adding new rows, updating rows, or deleting rows, in all relevant indexes with no additional action by users.

Retrieval performance of indexed data remains almost constant, even as new rows are inserted. However, the presence of many indexes on a table decreases the performance of updates, deletes, and inserts, because Oracle must also update the indexes associated with the table.

The optimizer can use an existing index to build another index. This results in a much faster index build.

Unique and Nonunique Indexes

Indexes can be unique or nonunique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column (or columns). Nonunique indexes do not impose this restriction on the column values.

Oracle recommends that unique indexes be created explicitly, and not through enabling a unique constraint on a table.

Alternatively, you can define UNIQUE integrity constraints on the desired columns. Oracle enforces UNIQUE integrity constraints by automatically defining a unique index on the unique key. However, it is advisable that any index that exists for query performance, including unique indexes, be created explicitly.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about creating unique indexes explicitly

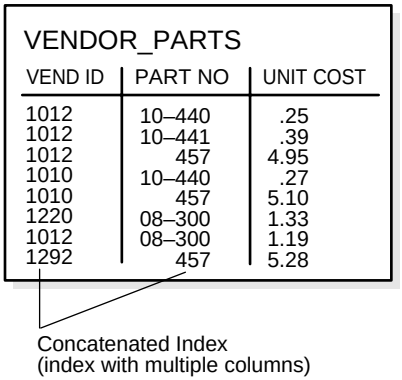
Composite Indexes

A **composite index** (also called a **concatenated index**) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order 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. Generally, the most commonly accessed or most selective columns go first.

Figure 11–6 illustrates the `VENDOR_PARTS` table that has a composite index on the `VENDOR_ID` and `PART_NO` columns.

Figure 11–6 Composite Index Example



No more than 32 columns can form a regular composite index. For a bitmap index, the maximum number columns is 30. A key value cannot exceed roughly one-half (minus some overhead) the available data space in a data block.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for more information about using composite indexes

Indexes and Keys

Although the terms are often used interchangeably, there is a distinction between **indexes** and **keys**. **Indexes** are structures actually stored in the database, which users create, alter, and drop using SQL statements. You create an index to provide a fast access path to table data. **Keys** are strictly a logical concept. Keys correspond to another feature of Oracle called integrity constraints, which enforce the business rules of a database.

Because Oracle uses indexes to enforce some integrity constraints, the terms key and index are often are used interchangeably. However, do not confuse them with each other.

> **See Also:** Chapter 23, "Data Integrity"

Indexes and Nulls

`NULL` values in indexes are considered to be distinct except when all the non-`NULL` values in two or more rows of an index are identical, in which case the rows are considered to be identical. Therefore, `UNIQUE` indexes prevent rows containing

NULL values from being treated as identical. This does not apply if there are no non-NULL values—in other words, if the rows are entirely NULL.

Oracle does not index table rows in which all key columns are NULL, except in the case of bitmap indexes or when the cluster key column value is null.

See Also: "Bitmap Indexes and Nulls" on page 11-51

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** precomputes the value of the function or expression and stores it in the index. You can create a function-based index as 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 PL/SQL function, package function, C callout, or SQL function. The expression cannot contain any aggregate functions, and it must be DETERMINISTIC. For building an index on a column containing an object type, the function can be a method of that object, such as a map method. However, you cannot build a function-based index on a LOB column, REF, or nested table column, nor can you build a function-based index if the object type contains a LOB, REF, or nested table.

See Also:

- "Bitmap Indexes"
- *Oracle9i Database Performance Guide and Reference* for more information about using function-based indexes

Uses of Function-Based Indexes

Function-based indexes provide an efficient mechanism for evaluating statements that contain functions in their WHERE clauses. The value of the expression is computed and stored in the index. When it processes INSERT and UPDATE statements, however, Oracle must still evaluate the function to process the statement.

For example, if you create the following index:

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);
```

then Oracle can use it when processing queries such as this:

```
SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;
```

Function-based indexes defined on `UPPER(column_name)` or `LOWER(column_name)` can facilitate case-insensitive searches. For example, the following index:

```
CREATE INDEX uppercase_idx ON emp (UPPER(empname));
```

can facilitate processing queries such as this:

```
SELECT * FROM emp WHERE UPPER(empname) = 'RICHARD';
```

A function-based index can also be used for a Globalization Support sort index that provides efficient linguistic collation in SQL statements.

> **See Also:** *Oracle9i Globalization Support Guide* for information about Globalization Support sort indexes

Optimization with Function-Based Indexes

You must gather statistics about function-based indexes for the optimizer. Otherwise, the indexes cannot be used to process SQL statements. Rule-based optimization never uses function-based indexes.

Cost-based optimization can use an index range scan on a function-based index for queries with expressions in `WHERE` clause. For example, in this query:

```
SELECT * FROM t WHERE a + b < 10;
```

the optimizer can use index range scan if an index is built on a+b. The range scan access path is especially beneficial when the predicate (`WHERE` clause) has low selectivity. In addition, the optimizer can estimate the selectivity of predicates involving expressions more accurately if the expressions are materialized in a function-based index.

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:** *Oracle9i Database Performance Guide and Reference* for more information about gathering statistics

Dependencies of Function-Based Indexes

Function-based indexes depend on the function used in the expression that defines the index. If the function is a PL/SQL function or package function, the index is disabled by any changes to the function specification.

PL/SQL functions used in defining function-based indexes must be `DETERMINISTIC`. The index owner needs the `EXECUTE` privilege on the defining function. If the `EXECUTE` privilege is revoked, then the function-based index is marked `DISABLED`.

See Also:

- *Oracle9i Database Performance Guide and Reference* for information about `DETERMINISTIC` PL/SQL functions
- "Function-Based Index Dependencies" on page 19-8 for more information about dependencies and privileges for function-based indexes

How Indexes Are Stored

When you create an index, Oracle automatically allocates an index segment to hold the index's data in a tablespace. You can control allocation of space for an index's segment and use of this reserved space in the following ways:

- Set the storage parameters for the index segment to control the allocation of the index segment's extents.
- Set the `PCTFREE` parameter for the index segment to control the free space in the data blocks that constitute the index segment's extents.

The tablespace of an index's segment is either the owner's default tablespace or a tablespace specifically named in the `CREATE INDEX` statement. You do not have to place an index in the same tablespace as its associated table. Furthermore, you can improve performance of queries that use an index by storing an index and its table in different tablespaces located on different disk drives, because Oracle can retrieve both index and table data in parallel.

See Also: "User Tablespace Settings and Quotas" on page 24-16

Format of Index Blocks

Space available for index data is the Oracle block size minus block overhead, entry overhead, rowid, and one length byte for each value indexed. The number of bytes required for the overhead of an index block depends on the operating system.

See Also: Your Oracle operating system specific documentation for information about the overhead of an index block

When you create an index, Oracle fetches and sorts the columns to be indexed and stores the rowid along with the index value for each row. Then Oracle loads the index from the bottom up. For example, consider the statement:

```
CREATE INDEX emp_ename ON emp(ename);
```

Oracle sorts the `EMP` table on the `ENAME` column. It then loads the index with the `ENAME` and corresponding rowid values in this sorted order. When it uses the index, Oracle does a quick search through the sorted `ENAME` values and then uses the associated rowid values to locate the rows having the sought `ENAME` value.

Although Oracle accepts the keywords `ASC, DESC, COMPRESS,` and `NOCOMPRESS` in the `CREATE INDEX` statement, they have no effect on index data, which is stored using rear compression in the branch nodes but not in the leaf nodes.

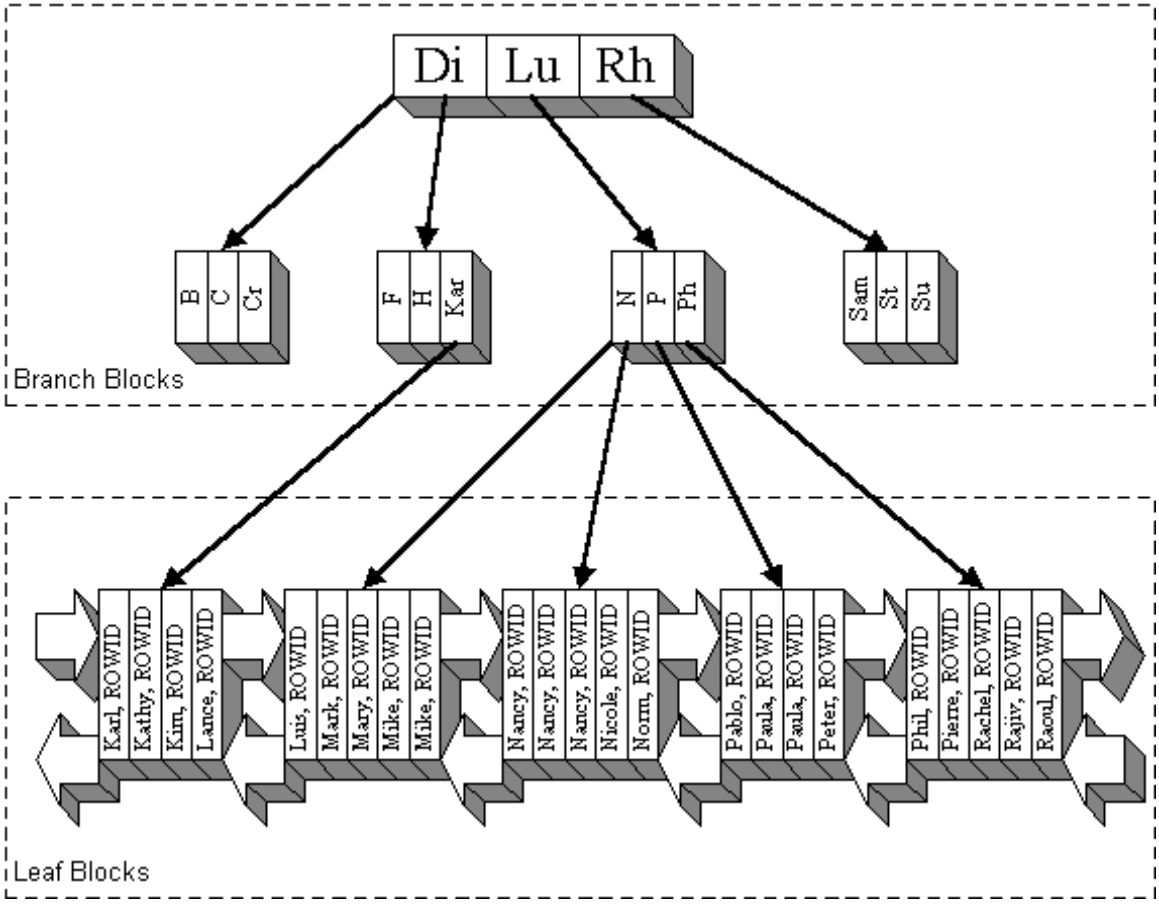
The Internal Structure of Indexes

Oracle uses B Trees to store indexes to speed up data access. If there are no indexes then you have to do a sequential scan on the data to find a value. For *n* rows, the average number of rows searched will be *n*/2. Obviously this does not scale very well as data volumes increase.

If we had an ordered list of the values, then we could divide it into block wide ranges (leaf blocks). The end points of the ranges along with pointers to the blocks could be stored in a search tree and we could find a value in log(*n*) time for *n* entries. This is the basic principle behind Oracle indexes.

Figure 11–7 illustrates the structure of a B-tree index.

Figure 11–7 Internal Structure of a B-tree Index



The upper blocks (**branch blocks**) of a B-tree index contain index data that points to lower-level index blocks. The lowest level index blocks (**leaf blocks**) contain every indexed data value and a corresponding rowid used to locate the actual row. The leaf blocks are doubly linked. Indexes in columns containing character data are based on the binary values of the characters in the database character set.

For a unique index, there is one rowid for each data value. 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. Key values containing all nulls are not indexed, except for

cluster indexes. Two rows can both contain all nulls without violating a unique index.

Index Properties

There are two kinds of blocks

- Branch blocks for searching
- Leaf blocks that store the values

Branch Blocks Branch blocks store the following:

- The minimum key prefix needed to make a branching decision between two keys
- The pointer to the child block containing the key

If the blocks have *n* keys then they have *n*+1 pointers. The number of keys and pointers is limited by the block size.

Leaf Blocks All leaf blocks are at the same depth from the root branch block. Leaf blocks store the following:

- The complete key value for every row
- `ROWID`s of the table rows

All key and `ROWID` pairs are linked to their left and right siblings. They are sorted by (key, `ROWID`).

Advantages of B-tree Structure

The B-tree structure has the following advantages:

- All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.
- B-tree indexes automatically stay balanced.
- All blocks of the B-tree are three-quarters full on the average.
- B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.

- B-tree performance is good for both small and large tables and does not degrade as the size of a table grows.

  See Also: Computer science texts for more information about B-tree indexes.

How Indexes Are Searched

Index Unique Scan

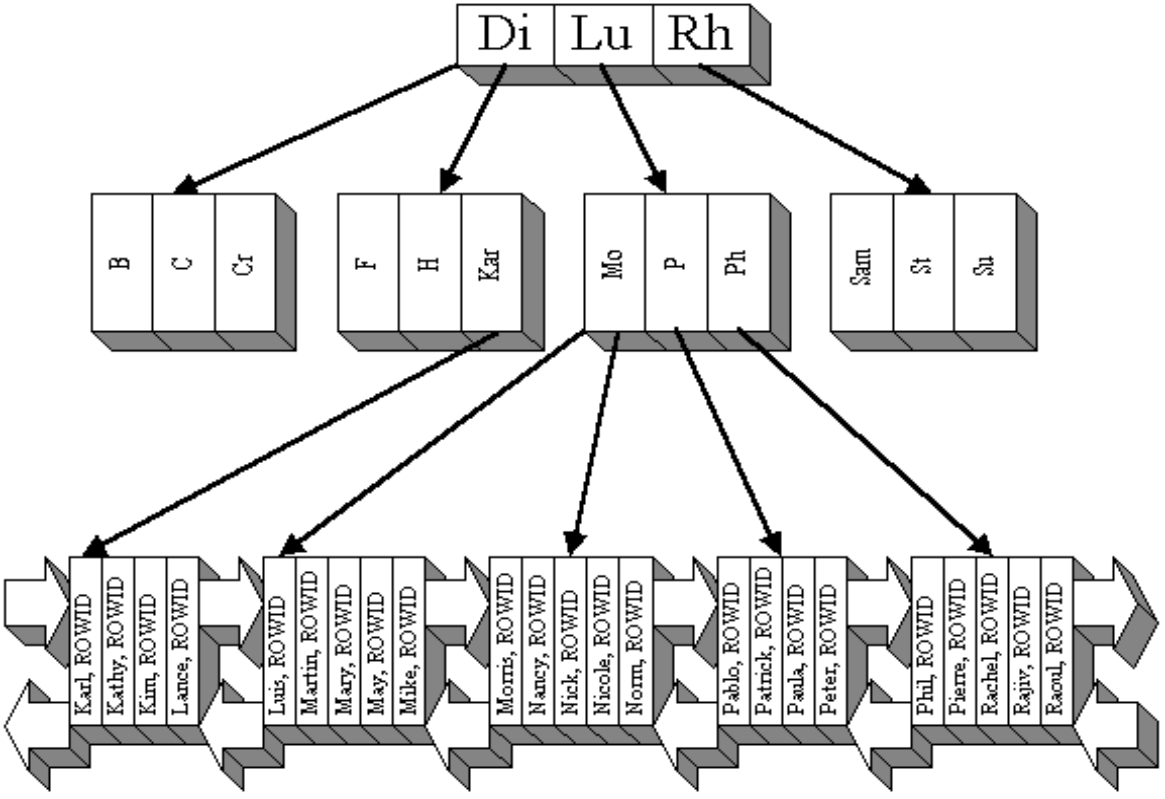
Index unique scan is one of the most efficient ways of accessing data. This access method is used for returning the data from B-tree indexes. The Optimizer chooses a unique scan when all columns of a unique (B-tree) index are specified with equality conditions.

Steps in Index Unique Scans

1. Start with the root block.
2. Search the block keys for the smallest key greater than or equal to the value.
3. If key > value, then follow the link before this key to the child block.
4. If key = value, then follow this link to the child block.
5. If there is no key greater than or equal to the value in Step 2, then follow the link after the highest key in the block.
6. Repeat steps 2 through 4 if the child block is a branch block.
7. Search the leaf block for key equal to the value.
8. If key is found, then return the `ROWID`.
9. If key is not found, then the row does not exist.

Figure 11–8 shows an example of an index unique scan and is described in the text that follows the figure.

Figure 11–8 Example of an Index Unique Scan



If searching for Patrick:

- In the root block, Rh is the smallest key >= Patrick.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, Ph is the smallest key >= Patrick.
- Follow the link before Ph to leaf block (Pablo, Patrick, Paula, Peter).
- In this block, search for key Patrick = Patrick.
- Found Patrick = Patrick, return `(KEY, ROWID)`.

If searching for Meg:

- In the root block, Rh is the smallest key >= Meg.

- Follow the link before Rh to branch block (N, P, Ph).
- In this block, Mo is the smallest key >= Meg.
- Follow the link before Mo to leaf block (Luis,… , May, Mike).
- In this block, search for key = Meg.
- Did not find key = Meg, return 0 rows.

Index Range Scan

Index range scan is a common operation for accessing selective data. It can be bounded (bounded on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted (in ascending order) by the ROWIDs.

How Index Range Scans Work Index range scans can happen on both unique and non-unique indexes. B-tree non-unique indexes are identical to the unique B-tree indexes. However, they allow multiple values for the same key.

For a range scan, you can specify an equality condition. For example:

- `name = 'ALEX' - start key = 'ALEX', end key = 'ALEX'`

Alternatively, specify an interval bounded by start key and end key. For example:

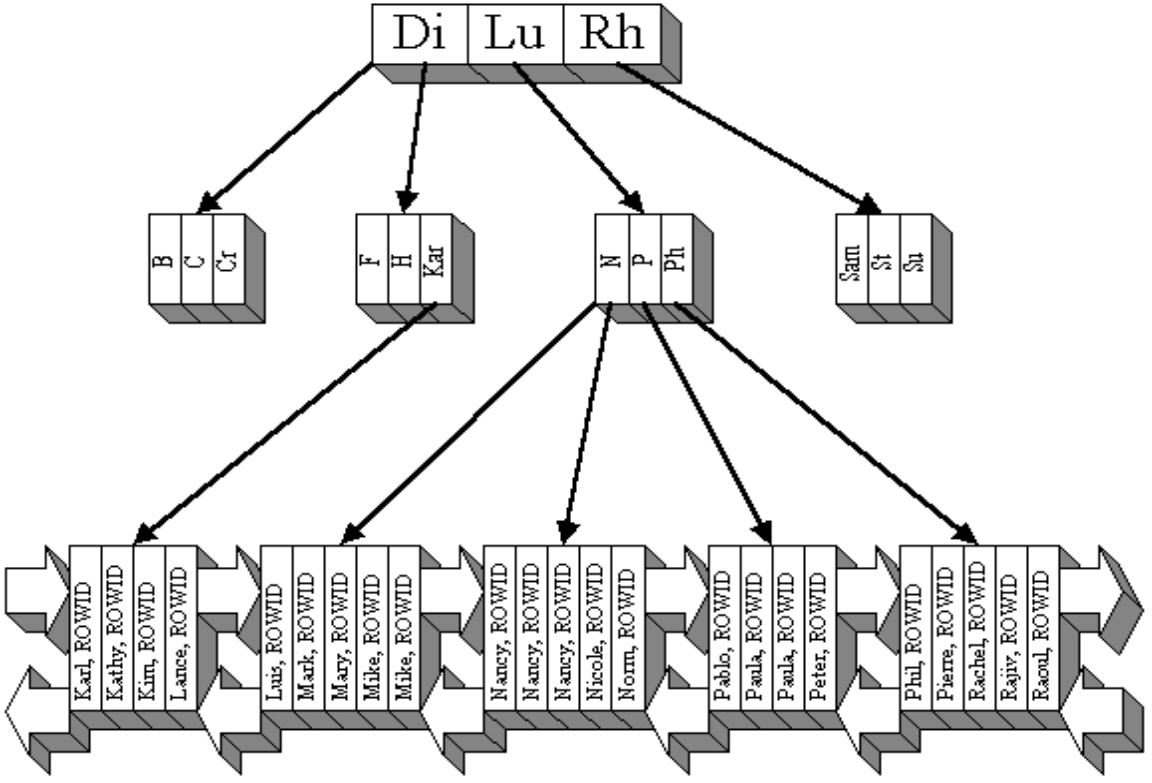
- `name LIKE 'AL%' - start key = 'AL, end key < 'AM'`
- `order_id BETWEEN 100 AND 120 - start key = 100, end key = 120`

Or, specify just a start key or an end key (unbounded range scan). For example:

- `order_book_date > SYSDATE - 30` (orders booked in last month)
- `employee_hire_date < SYSDATE - 3650` (employees with more than a decade of service)

Figure 11–9 shows an example of a bounded range scan and is described in the text that follows the figure.

Figure 11–9 Example of a Bounded Range Scan



Steps in a Bounded Range Scan

1. Start with the root block.
2. Search the block keys for the smallest key greater than or equal to the start key.
3. If key > start key, then follow the link before this key to the child block.
4. If key = start key, then follow this link to the child block.
5. If there is no key greater than or equal to the start key in Step 2, then follow the link after the highest key in the block.
6. Repeat steps 2 through 4 if the child block is a branch block.
7. Search the leaf block keys for the smallest key greater than or equal to the start key.

8. While key <= end key:
   - If the key columns meet all `WHERE` clause conditions, then return the (value, `ROWID`).
   - Follow the link to the right.

Here, the range scans make use of the fact that all the leaf nodes are linked from left to right. In Step 7, extra filtering conditions on the index columns can be applied before accessing the table by `ROWID`.

Range scans bounded on the left (unbounded on the right) start the same as above. However, they do not check for the end point. They continue until they reach the right-most leaf key.

Range scans bounded on the right traverse the index tree to the left-most leaf key and then follow step #6 and # 7 until they reach a key greater then the specified condition.

With range scans using the non-unique B-tree index, if searching for Nancy:

- Start key = 'Nancy', end key < 'Nancy'.
- In the root block, Rh is the smallest key >= start key.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, P is the smallest key >= start key.
- Follow the link before P to leaf block (Nancy, …, Nicole, Norm).
- In this block, Nancy is the smallest key >= start key.
- Because Nancy <= end key, return the (`KEY`, `ROWID`).
- Next key Nancy <= end key, return the (`KEY`, `ROWID`).
- Next key Nancy <= end key, return the (`KEY`, `ROWID`).
- Next key Nicole > end key, terminate the range scan.

If searching for 'P%':

- Start key = 'P', end key < 'Q'.
- In the root block, Rh is the smallest key >= start key.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, P is the smallest key = start key.
- Follow this link to leaf block (Pablo,…, Peter).

- In this block, Pablo is the smallest key >= start key.
- Because Pablo <= end key, return the (KEY, ROWID).
- Next key Paula <= end key, return the (KEY, ROWID).
- Next key Paula <= end key, return the (KEY, ROWID).
- Next key Phil <= end key, return the (KEY, ROWID).
- Next key Pierre <= end key, return the (KEY, ROWID).
- Next key Rachel > end key, terminate the range scan.

Index Range Scan Descending

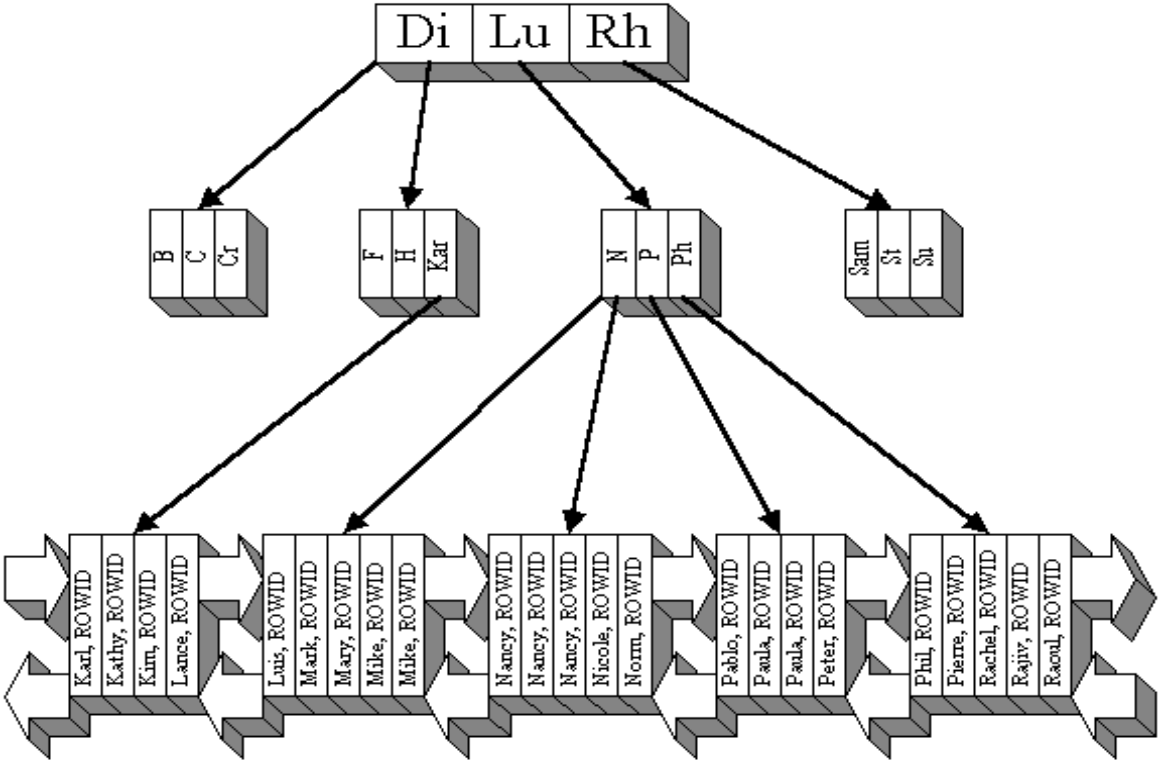
Steps in a Bounded Descending Range Scan For a descending range scan (like with the normal range scan), specify an equality condition or an interval.

1. Start with the root block.
2. Search the block keys for the biggest key less than or equal to the end key.
3. Follow the link to the child block.
4. If there is no key less than or equal to the end key in step 2, then follow the link before the lowest key in the block.
5. Repeat steps 2 through 4 if the child block is a branch block.
6. Search the leaf block keys for the biggest key less than or equal to the end key.
7. While key >= start key:
   - If the key columns meet all `WHERE` clause conditions, then return the (value, `ROWID`).
   - Follow the link to the left.

Here, the range scans make use of the fact that all the leaf nodes are linked from right to left.

Figure 11–10 shows examples of a bounded range scan and is described in the text that follows the figure.

Figure 11–10 Examples of Range Scans Using the Non-Unique B-tree Index



If searching for Nancy:

- Start key = 'Nancy', end key < 'Nancy'.
- In the root block, Lu is the biggest key <= end key.
- Follow the link to branch block (N, P, Ph).
- In this branch block, N is the biggest key <= end key.
- Follow the link after N to leaf block (Nancy, …, Nicole, Norm).
- In this leaf block, Nancy is the biggest key <= end key.
- Nancy >= start key, return the (`KEY`, `ROWID`).
- Prev key Nancy >= start key, return the (`KEY`, `ROWID`).

- Prev key Nancy >= start key, return the (KEY, ROWID).
- Prev key Mike < start key, terminate the range scan.

If searching for 'P%':

- Start key = 'P', end key < 'Q'.
- In the root block key, Lu is the biggest key <= end key.
- Follow the link to branch block (N, P, Ph).
- In this branch block, Ph is the biggest key <= end key.
- Follow the link to leaf block (Phil,…,Raoul).
- In the leaf block, Pierre is the biggest key <= end key.
- Pierre >= start key, return the (KEY, ROWID).
- Prev key Phil >= start key, return the (KEY, ROWID).
- Prev key Peter >= start key, return the (KEY, ROWID).
- Prev key Paula >= start key, return the (KEY, ROWID).
- Prev key Pablo >= start key, return the (KEY, ROWID).
- Prev key Norm < start key, terminate the range scan.

Key Compression

Key compression lets you compress portions of the primary key column values in an index or index-organized table, which reduces the storage overhead of repeated values.

Generally, keys in an index have two pieces, a grouping piece and a unique piece. If the key is not defined to have a unique piece, Oracle provides one in the form of a rowid appended to the grouping piece. Key compression is a method of breaking off the grouping piece and storing it so it can be shared by multiple unique pieces.

Prefix and Suffix Entries

Key compression breaks the index key into a prefix entry (the grouping piece) and a suffix entry (the unique piece). Compression is achieved by sharing the prefix entries among the suffix entries in an index block. 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.

Key compression is done within an index block but not across multiple index blocks. 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.

By default, the prefix consists of all key columns excluding the last one. For example, in a key made up of three columns (column1, column2, column3) the default prefix is (column1, column2). For a list of values (1,2,3), (1,2,4), (1,2,7), (1,3,5), (1,3,4), (1,4,4) the repeated occurrences of (1,2), (1,3) in the prefix are compressed.

Alternatively, you can specify the prefix length, which is the number of columns in the prefix. For example, if you specify prefix length 1, then the prefix is column1 and the suffix is (column2, column3). For the list of values (1,2,3), (1,2,4), (1,2,7), (1,3,5), (1,3,4), (1,4,4) the repeated occurrences of 1 in the prefix are compressed.

The maximum prefix length for a nonunique index is the number of key columns, and the maximum prefix length for a unique index is the number of key columns minus one.

Prefix entries are written to the index block only if the index block does not already contain a prefix entry whose value is equal to the present prefix entry. Prefix entries are available for sharing immediately after being written to the index block and remain available until the last deleted referencing suffix entry is cleaned out of the index block.

Performance and Storage Considerations

Key compression can lead to a huge saving in space, letting you store more keys in each index block, which can lead to less I/O and better performance.

Although key compression reduces the storage requirements of an index, it can increase the CPU time required to reconstruct the key column values during an index scan. It also incurs some additional storage overhead, because every prefix entry has an overhead of 4 bytes associated with it.

Uses of Key Compression

Key compression is useful in many different scenarios, such as:

- In a nonunique regular index, Oracle stores duplicate keys with the rowid appended to the key to break the duplicate rows. If key compression is used, Oracle stores the duplicate key as a prefix entry on the index block without the rowid. The rest of the rows are suffix entries that consist of only the rowid.

- This same behavior can be seen in a unique index that has a key of the form (**item**, **time stamp**), for example `(stock_ticker, transaction_time)`. Thousands of rows can have the same `stock_ticker` value, with `transaction_time` preserving uniqueness. On a particular index block a `stock_ticker` value is stored only once as a prefix entry. Other entries on the index block are `transaction_time` values stored as suffix entries that reference the common `stock_ticker` prefix entry.
- In an index-organized table that contains a `VARRAY` or `NESTED TABLE` datatype, the object ID (OID) is repeated for each element of the collection datatype. Key compression lets you compress the repeating OID values.

In some cases, however, key compression cannot be used. For example, in a unique index with a single attribute key, key compression is not possible, because there is a unique piece, but there are no grouping pieces to share.

See Also: "Index-Organized Tables" on page 11-57

Reverse Key Indexes

Creating a **reverse key index**, *c*ompared to a standard index, reverses the bytes of each column indexed (except the rowid) while keeping the column order. Such an arrangement can help avoid performance degradation with Oracle9*i* Real Application Clusters where modifications to the index are concentrated on a small set of leaf blocks. By reversing the keys of the index, the insertions become distributed across all leaf keys in the index.

Using the reverse key arrangement eliminates the ability to run an index range scanning query on the index. Because lexically adjacent keys are not stored next to each other in a reverse-key index, only fetch-by-key or full-index (table) scans can be performed.

Sometimes, using a reverse-key index can make an OLTP Oracle9*i* Real Application Clusters application faster. For example, keeping the index of mail messages in an e-mail application: some users keep old messages, and the index must maintain pointers to these as well as to the most recent.

The `REVERSE` keyword provides a simple mechanism for creating a reverse key index. You can specify the keyword `REVERSE` along with the optional index specifications in a `CREATE INDEX` statement:

```
CREATE INDEX i ON t (a,b,c) REVERSE;
```

You can specify the keyword `NOREVERSE` to `REBUILD` a reverse-key index into one that is not reverse keyed:

```
ALTER INDEX i REBUILD NOREVERSE;
```

Rebuilding a reverse-key index without the `NOREVERSE` keyword produces a rebuilt, reverse-key index. You cannot rebuild a normal index as a reverse key index. You must use the `CREATE` statement instead.

Bitmap Indexes

Note: Bitmap indexes are available only if you have purchased the Oracle9*i* Enterprise Edition.

See *Oracle9i Database New Features* for more information about the features available in Oracle9*i* and the Oracle9*i* Enterprise Edition.

The purpose of an index is to provide pointers to the rows in a table that contain a given key value. In a regular index, this is achieved by storing a list of rowids for each key corresponding to the rows with that key value. Oracle stores each key value repeatedly with each stored rowid. In a **bitmap index**, a bitmap for each key value is used instead of a list of rowids.

Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then it means that 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 regular index even though it uses a different representation internally. If the number of different key values is small, then bitmap indexes are very space efficient.

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 improves response time, often dramatically.

Benefits for Data Warehousing Applications

Bitmap indexing benefits data warehousing applications which have large amounts of data and ad hoc queries but a low level of concurrent transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries
- A substantial reduction of space usage compared to other indexing techniques
- Dramatic performance gains even on very low end hardware

- Very efficient parallel DML and loads

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space, because the index can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data. These indexes are primarily intended for decision support in data warehousing applications where users typically query the data rather than update it.

Bitmap indexes are also not suitable for columns that are primarily queried with less than or greater than comparisons. For example, a salary column that usually appears in `WHERE` clauses in a comparison to a certain value is better served with a B-tree index. Bitmapped indexes are only useful for `AND, OR, NOT,` or equality queries.

Bitmap indexes are integrated with the Oracle cost-based optimization approach and execution engine. They can be used seamlessly in combination with other Oracle execution methods. For example, the optimizer can decide to perform a hash join between two tables using a bitmap index on one table and a regular B-tree index on the other. The optimizer considers bitmap indexes and other available access methods, such as regular B-tree indexes and full table scan, and chooses the most efficient method, taking parallelism into account where appropriate.

Parallel query and parallel DML work with bitmap indexes as with traditional indexes. Bitmap indexes on partitioned tables must be local indexes. Parallel create index and concatenated indexes are also supported.

Cardinality

The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table. If the number of distinct values of a column is less than 1% of the number of rows in the table, or if the values in a column are repeated more than 100 times, then the column is a candidate for a bitmap index. Even columns with a lower number of repetitions and thus higher cardinality can be candidates if they tend to be involved in complex conditions in the `WHERE` clauses of queries.

For example, on a table with 1 million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can out-perform a B-tree index, particularly when this column is often queried in conjunction with other columns.

B-tree indexes are most effective for high-cardinality data: that is, data with many possible values, such as CUSTOMER_NAME or PHONE_NUMBER. In some situations, a B-tree index can be larger than the indexed data. Used appropriately, bitmap indexes can be significantly smaller than a corresponding B-tree index.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be quickly resolved by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered very quickly without resorting to a full table scan of the table.

Bitmap Index Example

Table 11–1 shows a portion of a company's customer data.

Table 11–1 Bitmap Index Example

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

MARITAL_STATUS, REGION, GENDER, and INCOME_LEVEL are all low-cardinality columns. There are only three possible values for marital status and region, two possible values for gender, and four for income level. Therefore, it is appropriate to create bitmap indexes on these columns. A bitmap index should not be created on CUSTOMER# because this is a high-cardinality column. Instead, use a unique B-tree index on this column to provide the most efficient representation and retrieval.

Table 11–2 illustrates the bitmap index for the REGION column in this example. It consists of three separate bitmaps, one for each region.

Table 11–2 Sample Bitmap

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

Each entry or bit in the bitmap corresponds to a single row of the `CUSTOMER` table. The value of each bit depends upon the values of the corresponding row in the table. For instance, the bitmap `REGION='east'` contains a one as its first bit. This is because the region is `east` in the first row of the `CUSTOMER` table. The bitmap `REGION='east'` has a zero for its other bits because none of the other rows of the table contain `east` as their value for `REGION`.

An analyst investigating demographic trends of the company's customers can ask, "How many of our married customers live in the central or west regions?" This corresponds to the following SQL query:

```
SELECT COUNT(*) FROM CUSTOMER
    WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

Bitmap indexes can process this query with great efficiency by counting the number of ones in the resulting bitmap, as illustrated in Figure 11–11. To identify the specific customers who satisfy the criteria, the resulting bitmap can be used to access the table.

Figure 11–11 Executing a Query Using Bitmap Indexes

Bitmap Indexes and Nulls

Bitmap indexes include rows that have `NULL` values, unlike most other types of indexes. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function `COUNT`.

Bitmap Indexes and Nulls Example 1

```
SELECT COUNT(*) FROM EMP;
```

Any bitmap index can be used for this query because all table rows are indexed, including those that have `NULL` data. If nulls were not indexed, the optimizer could only use indexes on columns with `NOT NULL` constraints.

Bitmap Indexes and Nulls Example 2

```
SELECT COUNT(*) FROM EMP WHERE COMM IS NULL;
```

This query can be optimized with a bitmap index on `COMM`.

Bitmap Indexes and Nulls Example 3

```
SELECT COUNT(*) FROM CUSTOMER WHERE GENDER = 'M' AND STATE != 'CA';
```

This query can be answered by finding the bitmap for `GENDER = 'M'` and subtracting the bitmap for `STATE = 'CA'`. If `STATE` can contain null values

(that is, if it does not have a `NOT NULL` constraint), then the bitmaps for `STATE = 'NULL'` must also be subtracted from the result.

Bitmap Indexes on Partitioned Tables

Like other indexes, you can create bitmap indexes 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.

See Also:

- Chapter 12, "Partitioned Tables and Indexes" for information about partitioned tables and descriptions of local and global indexes
- *Oracle9i Database Performance Guide and Reference* for more information about using bitmap indexes

Bitmap Join Indexes

A join index is an index on one table that involves columns of one or more different tables through a join.

The bitmap join index, in its simplest form, is a bitmap index on a table `F` based on columns from table `D1,...,Dn,` where `Di` joins with `F` in a star or snowflake schema as described in "Creation of a Bitmap Join Index" on page 11-56. In the data warehousing environment, table `F` is usually a fact table, table `Di` is usually a dimension table, and the join condition is an equi-inner join between the primary key column(s) of the dimension tables and the foreign key column(s) in the fact table. For simplicity, from now on we call the table whose rowids are bitmapped the ***fact table***, and the other tables participating in the join of bitmap join index the ***dimension tables***.

The volume of data that must be joined can be reduced if join indexes are used as joins have already been precalculated. In addition, join indexes which contain multiple dimension tables can eliminate bitwise operations which are necessary in the star transformation with existing bitmap indexes. Finally, bitmap join indexes are much more efficient in storage than materialized join views which do not compress rowids of the fact tables.

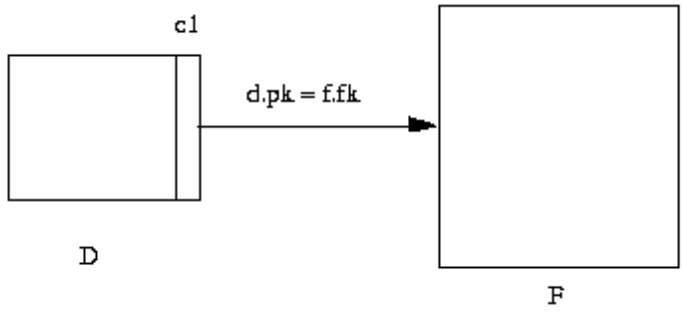
Four Join Models

The following is a description of four join models in the star query framework and how they are addressed by bitmap join indexes. The accompanying figures, 11–12 through 11–15, are described by SQL statements in the text that follows each figure.

Notation:

`F`*`i`* -- Fact table *i*
`D`*`i`* -- Dimension table *i*
`pk` -- The primary key column on the dimension table
`fk` -- The fact table column participating in the join with the dimension tables
`sales` -- The measurement column on the fact table

Figure 11–12 Model 1. One dimension table column joins with one fact table



The model shown in Figure 11–12, a bitmap join index on `F(D.c1),` can be represented by the following SQL statement:

```
CREATE BITMAP INDEX bji ON f (d.c1) FROM f, d WHERE d.pk = f.fk
```

Then the following query

```
select sum(f.sales)
from d, f
where d.pk = f.fk and d.c1 = 2
```

can be executed by accessing the bitmap join index to avoid the join operation.

Similar to the materialized join view, a bitmap join index precomputes the join and stores it as a database object. The difference is that a materialized join view materializes the join into a table while a bitmap join index materializes the join into a bitmap index.

Figure 11–13 Model 2. Two or more dimension table columns join with one fact table

.

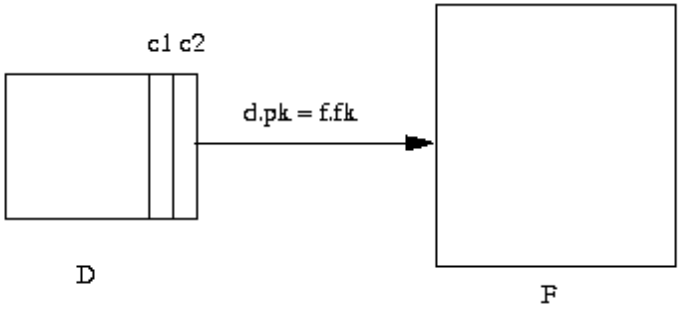


Figure 11–13 shows a simple extension of Model 1, requiring a concatenated bitmap join index to represent it, as follows:

```
CREATE BITMAP INDEX bji ON f (d.c1, d.c2) FROM F, d WHERE d.pk = f.fk;
```

The result of the following query:

```
select sum(f.sales)
from d, f
where d.pk = f.fk and d.c1 = 1 and d.c2 = 3;
```

can be retrieved by accessing the bitmap join index bji.

Another query which references only the leading portion of the index key can also use bitmap join index *bji*:

```
select sum(f.sales)
from d, f
where d.pk = f.fk and d.c1 = 1
```

Figure 11–14 Model 3. Multiple dimension tables join with one fact table

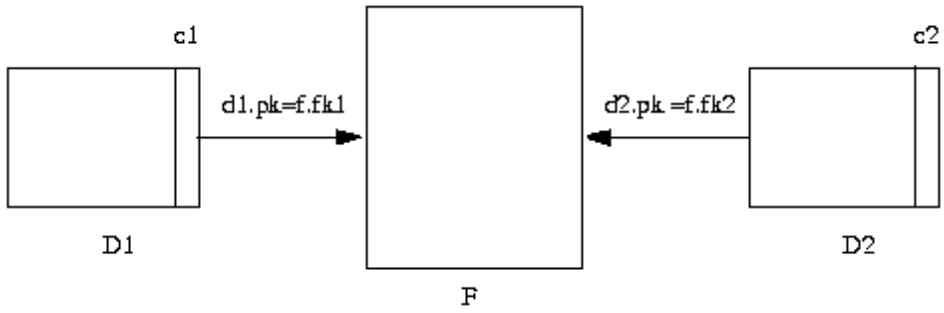


Figure 11–14 shows the third model, which requires a concatenated bitmap join index:

```
CREATE BITMAP INDEX bji ON f (d1.c1, d2.c2) FROM f, d1, d2
WHERE d1.pk = f.fk1 and d2.pk = f.fk2
```

Figure 11–15 Model 4. Snow Flake Schema

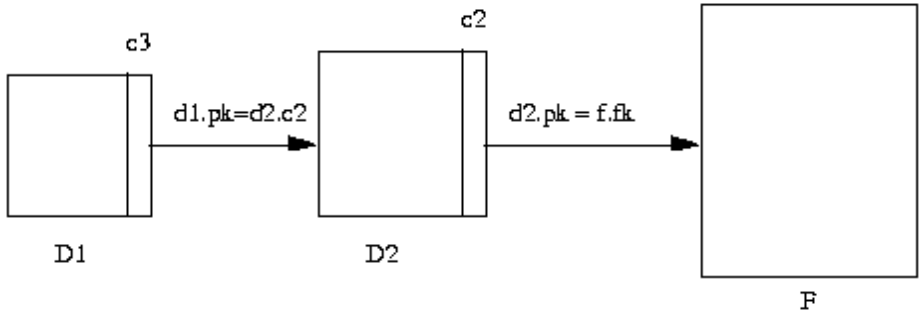


Figure 11–15 shows a model that involves joins between two or more dimension tables. It can be expressed by a bitmap join index. The bitmap join index can be either single or concatenated depending on the number of columns in the dimension tables to be indexed. A bitmap join index on `d1.c3` with a join between `d1` and `d2` and a join between `d2` and `f` can be created as follows:

```
CREATE BITMAP INDEX bji ON f (d1.c3) FROM f, d1, d2
WHERE d1.pk = d2.c2 and d2.pk = f.fk;
```

A bitmap join index should be able to represent joins of the combination of the above models.

Creation of a Bitmap Join Index

Consider a star or snowflake schema with a single fact table `F` and multiple dimension tables `D1,.., Dn` as defined in "Bitmap Join Indexes" on page 11-52. These are the restrictions on the bitmap join index on `F` joined with `D1,.., Dn.`

- The bitmap join index is on a single table `F.`
- No table can appear twice in the `FROM` clause.
- Joins form either star or snowflake schema and all joins are through primary keys or keys with unique constraints as follows:
  - The dimension table column(s) participating the join with the fact table must be either the primary key column(s) or with the unique constraint
  - In the snow flake schema where a join is `D1><D2><F,` the column(s) on `D1` participating in the join `D1><D2` must be either the primary key column(s) or with the unique constraint.
  - For a composite primary key on the dimension table, each column of the key needs to be in the join.
- All joins are equi-inner joins and they are connected by `AND`s only.
- The current restrictions for creating a regular bitmap index also apply to a bitmap join index. For example, we cannot create a bitmap index with the `UNIQUE` attribute. See the *Oracle9i SQL Reference* for other restrictions.
- A bitmap join index must not be partitioned if the fact table is not partitioned. If the fact table is partitioned, the corresponding bitmap join index must be local partitioned with the fact table. Global partitioned bitmap join indexes are not supported.

Bitmap join index on IOT, functional bitmap join index and temporary bitmap join index are not yet allowed.

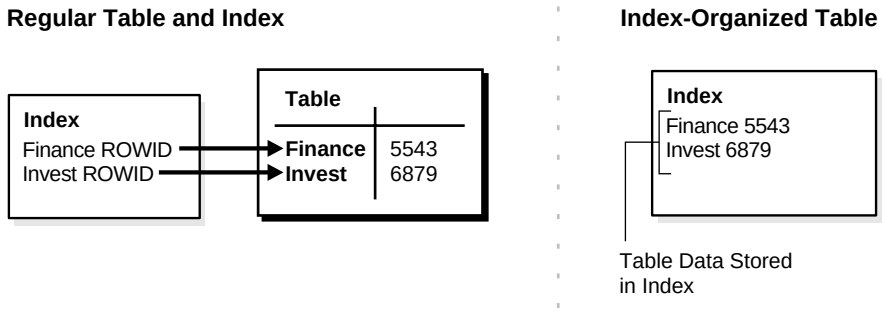
The primary key or unique constraint requirement is a correctness issue of a bitmap join index. For a regular bitmap index, there is a one-to-one mapping relation between a bit set in a bitmap and a rowid in the base table. For a bitmap join index, there should also be a one to one mapping between each row in the result set of the join and the rowids in the fact table. The primary key or unique constraint is used to enforce this one-to-one mapping.

Index-Organized Tables

An **index-organized table** has a storage organization that is a variant of a primary B-tree. Unlike an ordinary (heap-organized) table whose data is stored as an unordered collection (heap), data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner. Besides storing the primary key column values of an index-organized table row, each index entry in the B-tree stores the nonkey column values as well.

As shown in Figure 11–16, the index-organized table is somewhat similar to a configuration consisting of an ordinary table and an index on one or more of the table columns, but instead of maintaining two separate storage structures, one for the table and one for the B-tree index, the database system maintains only a single B-tree index. Also, rather than having a row's rowid stored in the index entry, the nonkey column values are stored. Thus, each B-tree index entry contains `<primary_key_value, non_primary_key_column_values>`.

Figure 11–16 Structure of a Regular Table Compared with an Index-Organized Table



Applications manipulate the index-organized table just like an ordinary table, using SQL statements. However, the database system performs all operations by manipulating the corresponding B-tree index.

Table 11–3 summarizes the differences between index-organized tables and ordinary tables.

Table 11–3 Comparison of Index-Organized Tables with Ordinary Tables

Ordinary Table	Index-Organized Table
Rowid uniquely identifies a row. Primary key can be optionally specified	Primary key uniquely identifies a row. Primary key must be specified
Physical rowid in `ROWID` pseudocolumn allows building secondary indexes	Logical rowid in `ROWID` pseudocolumn allows building secondary indexes
Access is based on rowid	Access is based on logical rowid
Sequential scan returns all rows	Full-index scan returns all rows
Can be stored in a cluster with other tables	Cannot be stored in a cluster
Can contain a column of the `LONG` datatype and columns of `LOB` datatypes	Can contain `LOB` columns but not `LONG` columns

Benefits of Index-Organized Tables

Index-organized tables provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Presence of nonkey columns of a row in the B-tree leaf block itself avoids an additional block access. Also, since rows are stored in primary key order, range access by the primary key (or a valid prefix) involves minimum block accesses.

In order to allow even faster access to frequently accessed columns, you can use a row overflow storage option (as described later) to push out infrequently accessed nonkey columns from the B-tree leaf block to an optional (heap-organized) overflow storage area. This allows limiting the size and content of the portion of a row that is actually stored in the B-tree leaf block, which may lead to a higher number of rows in each leaf block and a smaller B-tree.

Unlike a configuration of heap-organized table with a primary key index where primary key columns are stored both in the table and in the index, there is no such duplication here because primary key column values are stored only in the B-tree index.

Because rows are stored in primary key order, a significant amount of additional storage space savings can be obtained through the use of key compression.

Use of primary-key based logical rowids, as opposed to physical rowids, in secondary indexes on index-organized tables allows high availability. This is because, due to the logical nature of the rowids, secondary indexes do not become unusable even after a table reorganization operation that causes movement of the base table rows. At the same time, through the use of physical guess in the logical rowid, it is possible to get secondary index based index-organized table access

performance that is comparable to performance for secondary index based access to an ordinary table.

See Also:

- "Key Compression" on page 11-44
- "Secondary Indexes on Index-Organized Tables" on page 11-60
- *Oracle9i Database Administrator's Guide* for information about creating and maintaining index-organized tables

Index-Organized Tables with Row Overflow Area

B-tree index entries are usually quite small, because they only consist of the key value and a `ROWID`. In index-organized tables, however, the B-tree index entries can be large, because they consist of the entire row. This may destroy the dense clustering property of the B-tree index.

Oracle provides the `OVERFLOW` clause to handle this problem. You can specify an overflow tablespace so that, if necessary, a row can be divided into the following two parts that are then stored in the index and in the overflow storage area, respectively:

- The index entry, containing 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 nonkey columns, and
- The overflow part, containing column values for the remaining nonkey columns

With `OVERFLOW`, you can use two clauses, `PCTTHRESHOLD` and `INCLUDING`, to control how Oracle determines whether a row should be stored in two parts and if so, at which nonkey column to break the row. Using `PCTTHRESHOLD`, you can specify a threshold value as a percentage of the block size. If all the nonkey column values can be accommodated within the specified size limit, the row will not be broken into two parts. Otherwise, starting with the first nonkey column that cannot be accommodated, the rest of the nonkey columns are all stored in the row overflow storage area for the table.

The `INCLUDING` clause lets you specify a column name so that any nonkey column, appearing in the `CREATE TABLE` statement after that specified column, will be stored in the row overflow storage area. Please note that additional nonkey columns may sometimes need to be stored in the overflow due to `PCTTHRESHOLD`-based limits.

See Also: *Oracle9i Database Administrator's Guide* for examples of using the `OVERFLOW` clause

Secondary Indexes on Index-Organized Tables

Secondary index support on index-organized tables provides efficient access to index-organized table using columns that are not the primary key nor a prefix of the primary key.

Oracle constructs secondary indexes on index-organized tables using logical row identifiers (**logical rowids**) that are based on the table's primary key. A logical rowid optionally includes a **physical guess**, which identifies the block location of the row. Oracle can use these physical guesses to probe directly into the leaf block of the index-organized table, bypassing the primary key search. Because rows in index-organized tables do not have permanent physical addresses, the physical guesses can become stale when rows are moved to new blocks.

For an ordinary 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 secondary index scan followed by a scan of the primary key index.
- 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 physical guess), followed by a scan of the primary key index.

See Also: "Logical Rowids" on page 13-22

Bitmap Indexes on Index-Organized Tables

Oracle9*i*, Release 1 (9.0.1), supports bitmap indexes on index-organized tables. A mapping table is required for creating bitmap indexes on an index-organized table.

Mapping Table

The mapping table is a heap-organized table that stores logical rowids of the index-organized table. Specifically, each mapping table row stores one logical rowid for the corresponding index-organized table row. Thus, the mapping table provides

one-to-one mapping between logical rowids of the index-organized table rows and physical rowids of the mapping table rows.

A bitmap index on an index-organized table is similar to that on a heap-organized table except that the rowids used in the bitmap index on an index-organized table are those of the mapping table as opposed to the base table. There is one mapping table for each index-organized table and it is used by all the bitmap indexes created on that index-organized table.

In both heap-organized and index-organized base tables, a bitmap index is accessed using a search key. If the key is found, the bitmap entry is converted to a physical rowid. In the case of heap-organized table, this physical rowid is then used to access the base table. However, in the case of index-organized table, the physical rowid is then used to access the mapping table. The access to the mapping table yields a logical rowid. This logical rowid is used to access the index-organized table.

Though a bitmap index on an index-organized table does not store logical rowids, it is still logical in nature.

Note: Movement of rows in an index-organized table does not leave the bitmap indexes built on that index-organized table unusable. Movement of rows in the index-organized table does invalidate the physical guess in some of the mapping table's logical rowid entries. However, the index-organized table can still be accessed using the primary key.

Partitioned Index-Organized Tables

You can partition an index-organized table by `RANGE` or `HASH` on column values. The partitioning columns must form a subset of the primary key columns. Just like ordinary tables, local partitioned (prefixed and non-prefixed) index as well as global partitioned (prefixed) indexes are supported for partitioned index-organized tables.

B-tree Indexes on UROWID Columns for Heap- and Index-Organized Tables

`UROWID` datatype columns can hold logical primary key-based rowids identifying rows of index-organized tables. Oracle9*i*, Release 1 (9.0.1), supports indexes on `UROWID` datatypes of a heap- or index-organized table. The index supports equality predicates on `UROWID` columns. For predicates other than equality or for ordering on `UROWID` datatype columns, the index is not used.

Index-Organized Table Applications

The superior query performance for primary key based access, high availability aspects, and reduced storage requirements make index-organized tables ideal for the following kinds of applications:

- Online Transaction Processing (OLTP)
- Internet (for example, search engines and portals)
- E-Commerce (for example, electronic stores and catalogs)
- Data Warehousing
- Time-series applications

Application Domain Indexes

Oracle provides **extensible indexing** to accommodate indexes on customized complex data types such as documents, spatial data, images, and video clips and to make use of specialized indexing techniques. With extensible indexing, you can encapsulate application-specific index management routines as an **indextype** schema object and define a **domain index** (an application-specific index) on table columns or attributes of an object type. Extensible indexing also provides efficient processing of application-specific **operators**.

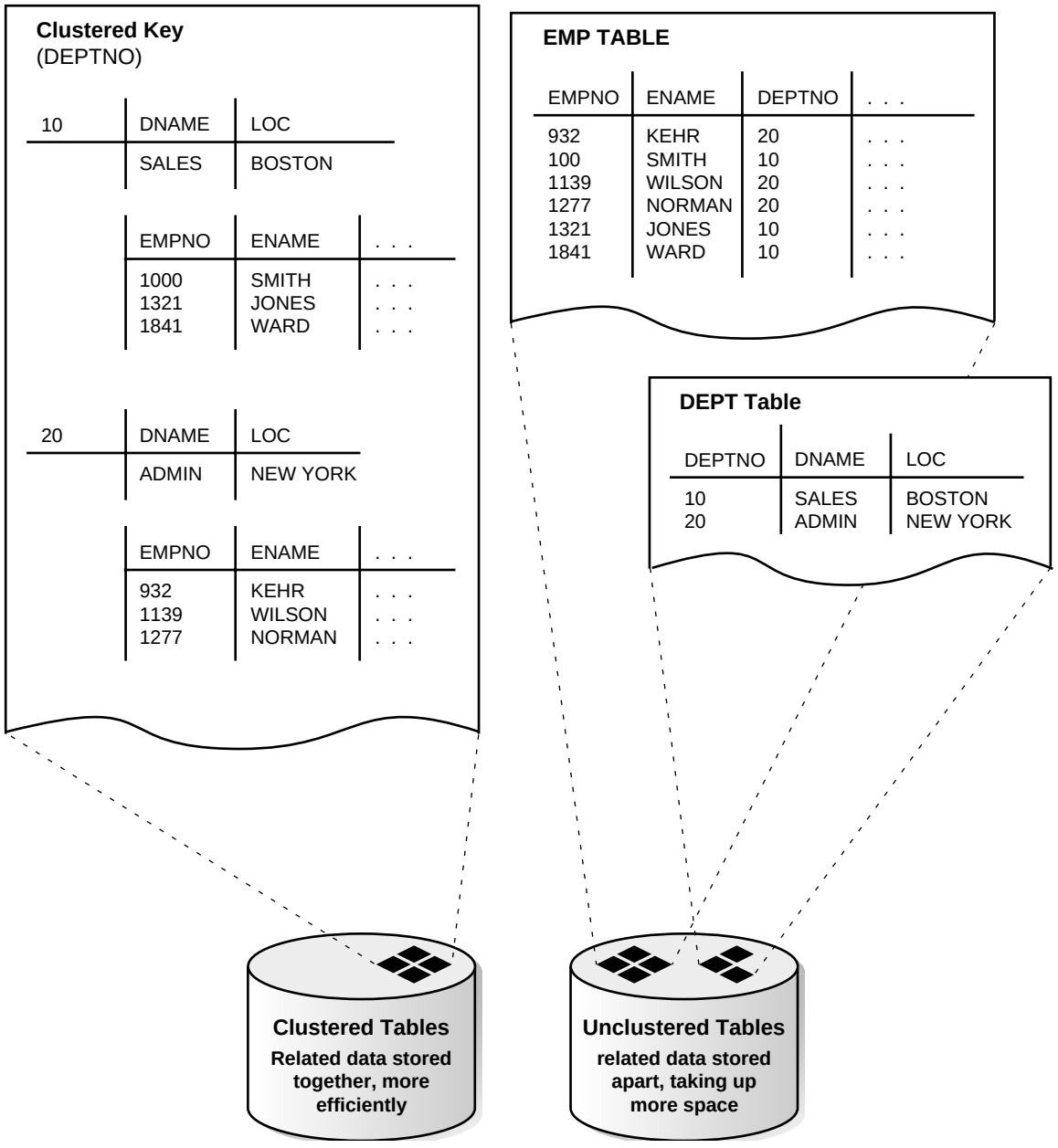
The application software, called the **cartridge**, controls the structure and content of a domain index. The Oracle server interacts with the application to build, maintain, and search the domain index. The index structure itself can be stored in the Oracle database as an index-organized table or externally as a file.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for information about using data cartridges within Oracle's extensibility architecture

Clusters

Clusters are an optional method of storing table data. A cluster is a group of tables that share the same data blocks because they share common columns and are often used together. For example, the `EMP` and `DEPT` table share the `DEPTNO` column. When you cluster the `EMP` and `DEPT` tables, Oracle physically stores all rows for each department from both the `EMP` and `DEPT` tables in the same data blocks. Figure 11–17 shows what happens when you cluster the `EMP` and `DEPT` tables:

Figure 11–17 Clustered Table Data



Because clusters store related rows of different tables together in the same data blocks, properly used clusters offers these benefits:

- Disk I/O is reduced for joins of clustered tables.
- Access time improves for joins of clustered tables.
- In a cluster, a **cluster key value** is the value of the cluster key columns for a particular row. Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value. Therefore, less storage is required to store related table and index data in a cluster than is necessary in nonclustered table format. For example, in Figure 11–17, notice how each cluster key (each `DEPTNO`) is stored just once for many rows that contain the same value in both the `EMP` and `DEPT` tables.

  See Also: *Oracle9i Database Administrator's Guide* for information about creating and managing clusters

Hash Clusters

Hashing is an optional way of storing table data to improve the performance of data retrieval. To use hashing, create a **hash cluster** and load tables into the cluster. Oracle physically stores the rows of a table in a hash cluster and retrieves them according to the results of a hash function.

Oracle uses a **hash function** to generate a distribution of numeric values, called **hash values**, which are based on specific cluster key values. The key of a hash cluster, like the key of an index cluster, can be a single column or composite key (multiple column key). To find or store a row in a hash cluster, Oracle applies the hash function to the row's cluster key value. The resulting hash value corresponds to a data block in the cluster, which Oracle then reads or writes on behalf of the issued statement.

A hash cluster is an alternative to a nonclustered table with an index or an index cluster. With an indexed table or index cluster, Oracle locates the rows in a table using key values that Oracle stores in a separate index.

To find or store a row in an indexed table or cluster, at least two I/Os must be performed:

- 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 cluster

In contrast, Oracle uses a hash function to locate a row in a hash cluster. No I/O is required. As a result, a minimum of one I/O operation is necessary to read or write a row in a hash cluster.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about creating and managing hash clusters

12

Partitioned Tables and Indexes

This chapter describes partitioned tables and indexes. It covers the following topics:

- Introduction to Partitioning
- Partitioning Methods
- Partitioned Indexes
- Partitioning to Improve Performance

> **Note:** Oracle supports partitioning only for tables, indexes on tables, materialized views, and indexes on materialized views. Oracle does not support partitioning of clustered tables or indexes on clustered tables.

Introduction to Partitioning

Partitioning addresses key issues in supporting very large tables and indexes by letting you decompose them into smaller and more manageable pieces called **partitions**. SQL queries and DML statements do not need to be modified in order to access partitioned tables. However, once partitions are defined, DDL statements can access and manipulate individuals partitions rather than entire tables or indexes. This is how partitioning can simplify the manageability of large database objects. Also, partitioning is entirely transparent to applications.

Each partition of a table or index must have the same logical attributes, such as column names, datatypes, and constraints, but each partition can have separate physical attributes such as pctfree, pctused, and tablespaces.

Partitioning is useful for many different types of applications, particularly applications that manage large volumes of data. OLTP systems often benefit from improvements in manageability and availability, while data warehousing systems benefit from performance and manageability.

Note: All partitions of a partitioned object must reside in tablespaces of a single block size.

See Also:

- "Nonstandard Block Sizes" on page 4-13
- *Oracle9i Data Warehousing Guide* for more information about partitioning

Partitioning offers these advantages:

- Partitioning enables data management operations such data loads, index creation and rebuilding, and backup / recovery at the partition level, rather than on the entire table. This results in significantly reduced times for these operations.
- Partitioning improves query performance. In many cases, the results of a query can be achieved by accessing a subset of partitions, rather than the entire table. For some queries, this technique (called **partition pruning**) can provide order-of-magnitude gains in performance.
- Partitioning can significantly reduce the impact of scheduled downtime for maintenance operations.

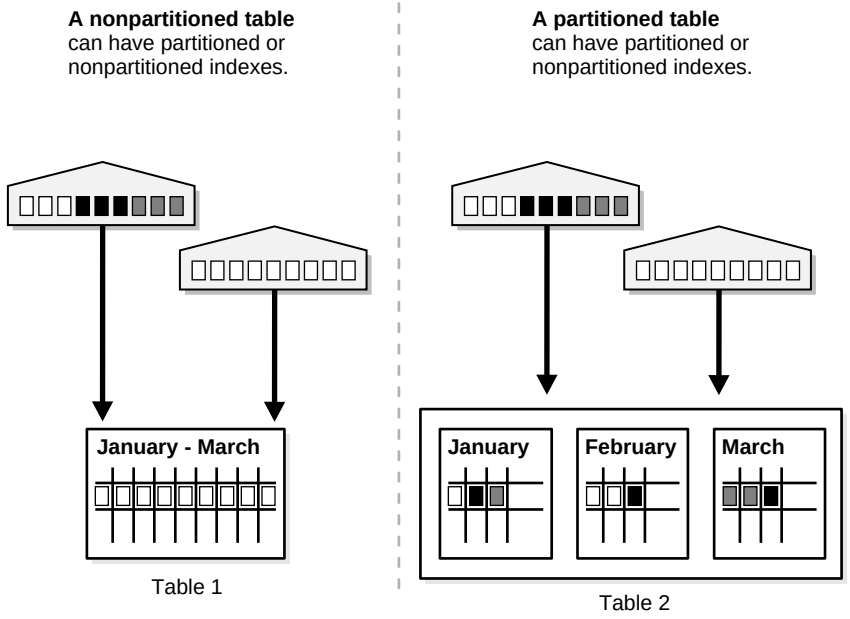
Partition independence for partition maintenance operations lets you perform concurrent maintenance operations on different partitions of the same table or index. You can also run concurrent SELECT and DML operations against partitions that are unaffected by maintenance operations.

- Partitioning increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures.
- Partitioning can be implemented without requiring any modifications to your applications.

  For example, you could convert a nonpartitioned table to a partitioned table without needing to modify any of the SELECT statements or DML statements which access that table. You do not need to rewrite your application code to take advantage of partitioning.

Figure 12–1 offers a graphical view of how partitioned tables differ from nonpartitioned tables.

Figure 12–1 A View of Partitioned Tables



Partition Key

Each row in a partitioned table is unambiguously assigned to a single partition. The partition key is a set of one or more columns that determines the partition for each row. Oracle9*i* automatically directs insert, update, and delete operations to the appropriate partition through the use of the partition key. A partition key:

- consists of an ordered list of 1 to 16 columns
- cannot contain a `LEVEL`, `ROWID`, or `MLSLABEL` pseudocolumn or a column of type `ROWID`
- can contain columns which all the `NULL` value

Partitioned Tables

Tables can be partitioned into any number of separate partitions. Any table can be partitioned except those tables containing columns with `LONG` or `LONG RAW` datatypes. You can, however, use tables containing columns with `CLOB` or `BLOB` datatypes.

Partitioned Index-Organized Tables

You can range partition index-organized tables. This feature is very useful for providing improved manageability, availability and performance for index-organized tables. In addition, data cartridges that use index-organized tables can take advantage of the ability to partition their stored data. Common examples of this are the Image and *inter*Media cartridges.

For partitioning an index-organized table:

- Only range and hash partitioning are supported
- 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

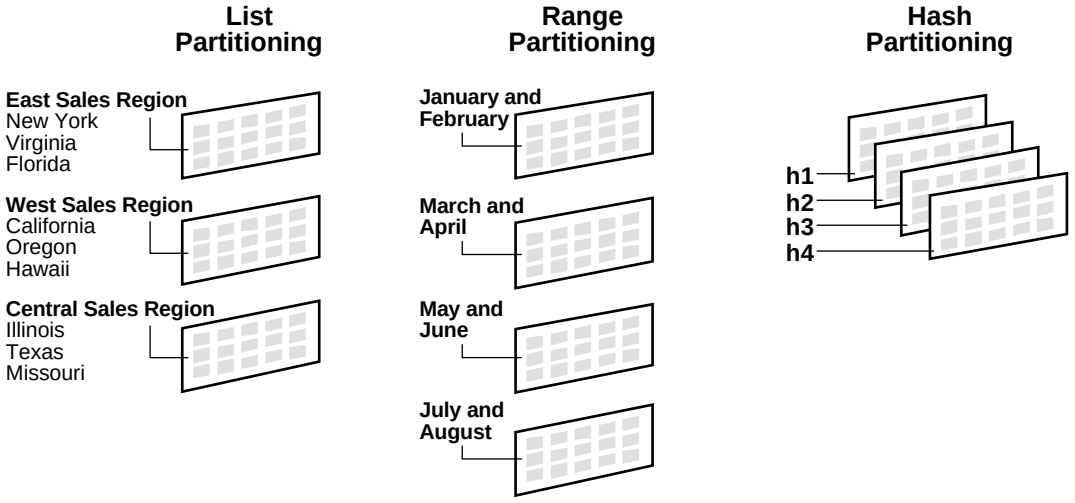
Partitioning Methods

Oracle provides the following partitioning methods:

- Range Partitioning
- List Partitioning
- Hash Partitioning
- Composite Partitioning

Figure 12–2 offers a graphical view of the methods of partitioning.

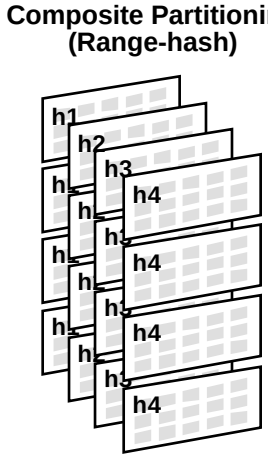
Figure 12–2 List, Range, and Hash Partitioning



Composite partitioning is a combination of other partitioning methods. Oracle currently supports range-hash composite partitioning. Figure 12–3 offers a graphical view of composite partitioning.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about partitioning methods

Figure 12–3 Composite Partitioning



Range Partitioning

Range partitioning maps data to partitions based on ranges of partition key values that you establish for each partition. It is the most common type of partitioning and is often used with dates. For example, you might want to partition sales data into monthly partitions.

When using range partitioning, there are a few rules to keep in mind:

- Each partition has a `VALUES LESS THAN` clause, which specifies a noninclusive upper bound for the partitions. Any binary values of the partition key equal to or higher than this literal are added to the next higher partition.
- All partitions, except the first, have an implicit lower bound specified by the `VALUES LESS THAN` clause on the previous partition.
- A `MAXVALUE` literal can be defined for the highest partition. `MAXVALUE` represents a virtual infinite value that sorts higher than any other possible value for the partition key, including the null value.

A typical example is given below. The statement creates a table `(SALES_RANGE)` that is range partitioned on the `SALES_DATE` field.

Range Partitioning Example

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY RANGE(sales_date)
(
PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY')),
)
```

List Partitioning

List partitioning enables you to explicitly control how rows map to partitions. You do this by specifying a list of discrete values for the partitioning key in the description for each partition. This is different from range partitioning, where a range of values is associated with a partition and from hash partitioning, where a hash function controls the row-to-partition mapping. The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way.

The details of list partitioning can best be described with an example. In this case, let's say you want to partition a sales table by region. That means grouping states together according to their geographical location as in the following example.

List Partitioning Example

```
CREATE TABLE sales_list
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_state   VARCHAR2(20),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY LIST(sales_state)
(
PARTITION sales_west VALUES IN('California', 'Hawaii'),
PARTITION sales_east VALUES IN ('New York', 'Virginia', 'Florida'),
PARTITION sales_central VALUES IN('Texas', 'Illinois'),
)
```

A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within the set of values that describes the partition. For example, the following rows will be inserted as follows:

- `(10, 'Jones', 'Hawaii', 100, '05-JAN-2000')` maps to partition `q1_west`
- `(21, 'Smith', 'Florida', 150, '15-JAN-2000')` maps to partition `q1_east`
- `(32, 'Lee', 'Colorado', 130, '21-JAN-2000')` will not map to any partition in the table

Unlike range and hash partitioning, multicolumn partition keys are not supported for list partitioning. If a table is partitioned by list, the partitioning key can only consist of a single column of the table.

Hash Partitioning

Hash partitioning enables easy partitioning of data that does not lend itself to range or list partitioning. It does this with a simple syntax and is easy to implement. It is a better choice than range partitioning when:

- You do not know beforehand how much data will map into a given range
- The sizes of range partitions would differ quite substantially or would be difficult to balance manually
- Range partitioning would cause the data to be undesirably clustered
- Performance features such as parallel DML, partition pruning, and partition-wise joins are important

The concepts of splitting, dropping or merging partitions do not apply to hash partitions. Instead, hash partitions can be added and coalesced.

Hash Partitioning Example

```
CREATE TABLE sales_hash
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
week_no       NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4
STORE IN (data1, data2, data3, data4)
```

The above statement creates a table SALES_HASH, which is hash partitioned on SALESMAN_ID field. Data1, data2, data3 and data4 are tablespace names.

Composite Partitioning

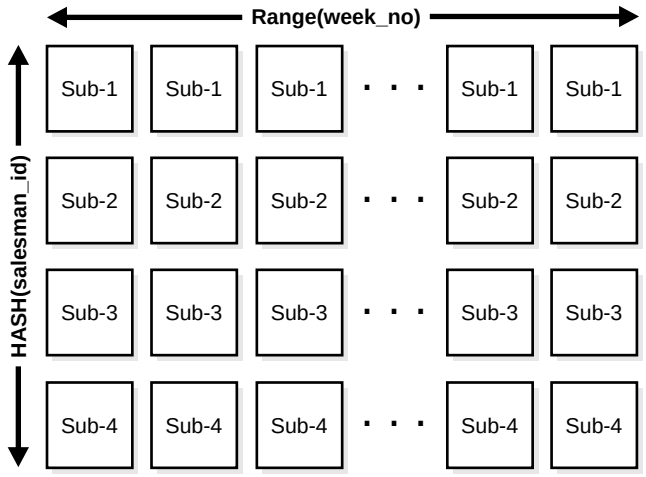
Composite partitioning partitions data using the range method, and within each partition, subpartitions it using the hash method. Composite partitioning provides the improved manageability of range partitioning and the data placement, striping, and parallelism advantages of hash partitioning. Composite partitioning supports historical operations, such as adding new range partitions, but also provides higher degrees of parallelism for DML operations and finer granularity of data placement through subpartitioning.

Composite Partitioning Example

```
CREATE TABLE sales_composite
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY RANGE(sales_date)
SUBPARTITION BY HASH(salesman_id)
SUBPARTITIONS 4
PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY'))
( SUBPARTITION sales_jan2000_1 TABLESPACE data1,
  SUBPARTITION sales_jan2000_2 TABLESPACE data2,
  SUBPARTITION sales_jan2000_3 TABLESPACE data3,
  SUBPARTITION sales_jan2000_4 TABLESPACE data4),
PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),(
SUBPARTITION sales_jan2000_1 TABLESPACE data1,
  SUBPARTITION sales_jan2000_2 TABLESPACE data2,
  SUBPARTITION sales_jan2000_3 TABLESPACE data3,
  SUBPARTITION sales_jan2000_4 TABLESPACE data4),
PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY'))
( SUBPARTITION sales_jan2000_1 TABLESPACE data1,
  SUBPARTITION sales_jan2000_2 TABLESPACE data2,
  SUBPARTITION sales_jan2000_3 TABLESPACE data3,
  SUBPARTITION sales_jan2000_4 TABLESPACE data4);
```

This statement creates a table SALES_COMP that is range partitioned on the SALES_DATE field and hash subpartitioned on SALESMAN_ID. SALES_COMP will have 16 individual subpartitions. Figure 12–4 offers a graphical view of the above example.

Figure 12–4 Composite Partitioning



When to Partition a Table

Here are some suggestions for when to partition a table:

- Tables greater than 2GB should always be considered for partitioning.
- Tables on which you want to perform parallel DML operations must be partitioned.
- Tables containing historical data, in which new data is added into the newest partition. A typical example is a historical table where only the current month's data is updatable and the other 11 months are read-only.

Partitioned Indexes

Just like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability. They can either be partitioned independently (global indexes) or automatically linked to a table's partitioning method (local indexes).

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about partitioned indexes

Local Partitioned Indexes

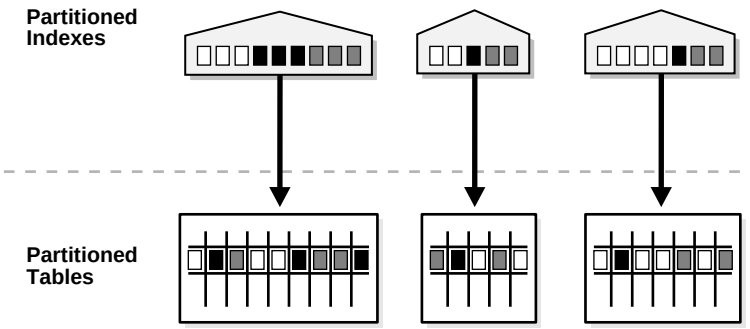
Local partitioned indexes are easier to manage than other types of partitioned indexes. They also offer greater availability and are common in DSS environments. The reason for this is equipartitioning: each partition of a local index is associated with exactly one partition of the table. This enables Oracle to automatically keep the index partitions in sync with the table partitions, and makes each table-index pair independent. Any actions that make one partition's data invalid or unavailable will only affect a single partition.

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.

A local index can be unique. However, in order for a local index to be unique, the partitioning key of the table must be part of the index's key columns. Unique local indexes are useful for OLTP environments.

Figure 12–5 offers a graphical view of local partitioned indexes.

Figure 12–5 Local Partitioned Index



Global Partitioned Indexes

Global partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method. They are commonly used for OLTP environments and offer efficient access to any individual record.

The highest partition of a global index must have a partition bound, all of whose values are MAXVALUE. This ensures that all rows in the underlying table can be represented in the index. Global prefixed indexes can be unique or nonunique.

You cannot add a partition to a global index because the highest partition always has a partition bound of MAXVALUE. If you wish to add a new highest partition, use the ALTER INDEX SPLIT PARTITION statement. If a global index partition is empty, you can explicitly drop it by issuing the ALTER INDEX DROP PARTITION statement. If a global index partition contains data, dropping the partition causes the next highest partition to be marked unusable. You cannot drop the highest partition in a global index.

Maintenance of Global Partitioned Indexes

By default, the following operations on partitions on a heap-organized table mark all global indexes as unusable:

```
ADD (HASH)
COALESCE (HASH)
DROP
EXCHANGE
MERGE
MOVE
SPLIT
TRUNCATE
```

In Oracle9*i*, Release 1 (9.0.1), these indexes can be maintained by appending the clause UPDATE GLOBAL INDEXES to the SQL statements for the operation. There are two advantages to maintaining global indexes:

- The index remains available and online throughout the operation. Hence no other applications are affected by this operation.
- The index doesn't have to be rebuilt after the operation.

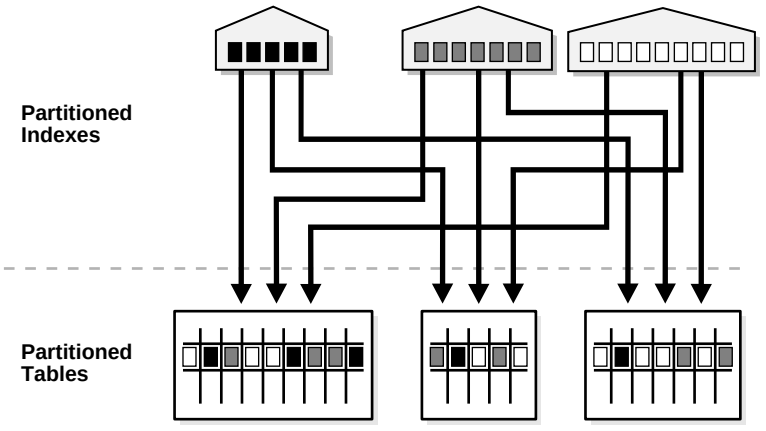
Example: `ALTER TABLE DROP PARTITION P1 UPDATE GLOBAL INDEXES`

Note: This feature is supported only for heap organized tables.

See Also: *Oracle9i SQL Reference* for more information about the UPDATE GLOBAL INDEX clause

Figure 12–6 offers a graphical view of global partitioned indexes.

Figure 12–6 Global Partitioned Index

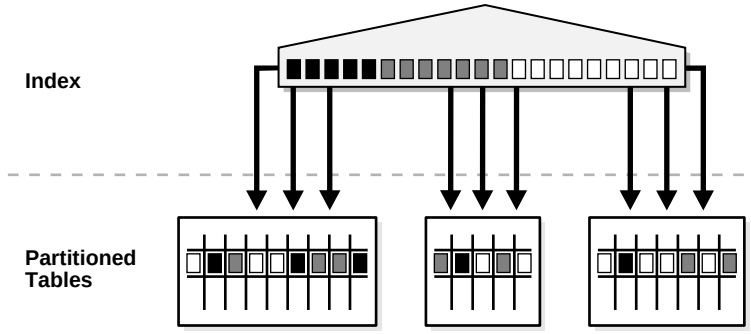


Global Nonpartitioned Indexes

Global nonpartitioned indexes behave just like a nonpartitioned index. They are commonly used in OLTP environments and offer efficient access to any individual record.

Figure 12–7 offers a graphical view of global nonpartitioned indexes.

Figure 12–7 Global Nonpartitioned Index



Index Examples

Example of Creation with EMP

```
CREATE TABLE emp
(empno  NUMBER(4) NOT NULL,
 ename  VARCHAR2(10),
 deptno NUMBER(2))
PARTITION BY RANGE (deptno)
(PARTITION emp_part1 VALUES LESS THAN (11) TABLESPACE part1,
 PARTITION emp_part2 VALUES LESS THAN (21) TABLESPACE part2,
 PARTITION emp_part3 VALUES LESS THAN (31) TABLESPACE part3)
```

Example of a Local Index on EMP

```
CREATE INDEX emp_local_idx ON emp (empno) LOCAL
```

Example of a Global Index

```
CREATE INDEX emp_global_idx ON emp(empno);
```

Example of a Global Partitioned Index

```
CREATE INDEX emp_glbbal_part_idx ON emp(empno)
GLOBAL PARTITION BY RANGE(empno)
(PARTITION p1 VALUES LESS THAN(5000),
 PARTITION p2 VALUES LESS THAN(MAXVALUE));
```

Example of a Partitioned Index-Organized Table

```
CREATE TABLE sales_range
(
salesman_id   NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_data    DATE,
PRIMARY KEY(sales_date, salesman_id))
ORGANIZATION INDEX INCLUDING salesman_id
OVERFLOW TABLESPACE tabsp_overflow
PARTITION BY RANGE(week_no)
(PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p1_overflow,
 PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p2_overflow,
 PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p3_overflow,
 PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p4_overflow)
```

Miscellaneous Information about Creating Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables, with the restriction that the bitmap indexes must be local to the partitioned table. They cannot be global indexes.

Global indexes can be unique. Local indexes can only be unique if the partitioning key is a part of the index key.

Using Partitioned Indexes in OLTP Applications

Here are a few guidelines for OLTP applications:

- Global indexes and unique, local indexes provide better performance than nonunique local indexes because they minimize the number of index partition probes.
- Local indexes offer better availability when there are partition or subpartition maintenance operations on the table.

Using Partitioned Indexes in DSS Applications

Here are a few guidelines for DSS applications:

- Local indexes are preferable because they are easier to manage during data loads and during partition-maintenance operations.
- Local indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

Partitioned Indexes on Composite Partitions

Here are a few points to remember when using partitioned indexes on composite partitions:

- Only range partitioned global indexes are supported.
- Subpartitioned indexes are always local and stored with the table subpartition by default.
- Tablespaces can be specified at either index or index subpartition levels.

Partitioning to Improve Performance

Partitioning can help you improve performance and manageability. Some topics to keep in mind when using partitioning for these reasons are:

- Partition Pruning
- Partition-wise Joins
- Parallel DML

Partition Pruning

The Oracle server explicitly recognizes partitions and subpartitions. It then optimizes SQL statements to mark the partitions or subpartitions that need to be accessed and eliminates (prunes) unnecessary partitions or subpartitions from access by those SQL statements. In other words, partition pruning is the skipping of unnecessary index and data partitions or subpartitions in a query.

For each SQL statement, depending on the selection criteria specified, unneeded partitions or subpartitions can be eliminated. For example, if a query only involves March sales data, there is no need to retrieve data for the remaining eleven months. Such intelligent pruning can dramatically reduce the data volume, resulting in substantial improvements in query performance.

If the optimizer determines that the selection criteria used for pruning are satisfied by all the rows in the accessed partition or subpartition, it removes those criteria

from the predicate list (WHERE clause) during evaluation in order to improve performance. However, the optimizer cannot prune partitions if the SQL statement applies a function to the partitioning column (with the exception of the TO_DATE function). Similarly, the optimizer cannot use an index if the SQL statement applies a function to the indexed column, unless it is a function-based index.

Pruning can eliminate index partitions even when the underlying table's partitions cannot be eliminated, but only when the index and table are partitioned on different columns. You can often improve the performance of operations on large tables by creating partitioned indexes that reduce the amount of data that your SQL statements need to access or modify.

Equality, inequality (<, >, between) and IN-list predicates are considered for partition pruning with range partitioning, and equality and IN-list predicates are considered for partition pruning with hash partitioning.

Partition Pruning Example

We have a partitioned table called ORDERS. The partition key for ORDERS is ORDER_DATE. Let's assume that ORDERS has 6 months of data, January to June, with a partition for each month of data. If the following query is executed

```
SELECT SUM(value) FROM orders
WHERE order_date BETWEEN '28-MAR-98' AND '23-APR-98'
```

Partition pruning is achieved by:

- First, partition elimination of January, February, May, and June data partitions. Then either:
  - An index scan of the March and April data partition due to high index selectivity

    or
  - A full scan of the March and April data partition due to low index selectivity

Partition-wise Joins

A partition-wise join is a join optimization that you can use when joining two tables that are both partitioned along the join column(s). With partition-wise joins, the join operation is broken into smaller joins that are performed sequentially or in parallel. Another way of looking at partition-wise joins is that they minimize the amount of

data exchanged among parallel slaves during the execution of parallel joins by taking into account data distribution.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about partitioning methods and partition-wise joins

Parallel DML

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems and data warehouses. In addition to conventional tables, you can use parallel query and parallel DML with range- and hash-partitioned tables. By doing so, you can enhance scalability and performance for batch operations.

The semantics and restrictions for parallel DML sessions are the same whether you are using index-organized tables or not.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about parallel DML and its use with partitioned tables

13

System-Provided Datatypes

This chapter discusses the Oracle built-in datatypes, their properties, and how they map to non-Oracle datatypes. Topics include:

- Introduction to Oracle Datatypes
- Character Datatypes
- NUMBER Datatype
- DATE Datatype
- LOB Datatypes
- RAW and LONG RAW Datatypes
- ROWID and UROWID Datatypes
- ANSI, DB2, and SQL/DS Datatypes
- XML Datatypes
- Data Conversion

Introduction to Oracle Datatypes

Each column value and constant in a SQL statement has a **datatype**, which is associated with a specific storage format, constraints, and a valid range of values. When you create a table, you must specify a datatype for each of its columns.

Oracle provides the following built-in datatypes:

- Character Datatypes
  - CHAR Datatype
  - VARCHAR2 and VARCHAR Datatypes
  - NCHAR and NVARCHAR2 Datatypes
  - LONG Datatype
- NUMBER Datatype
- DATE Datatype
- LOB Datatypes
  - BLOB Datatype
  - CLOB and NCLOB Datatypes
  - BFILE Datatype
- RAW and LONG RAW Datatypes
- ROWID and UROWID Datatypes
  - Physical Rowids
  - Logical Rowids
  - Rowids in Non-Oracle Databases

Note: PL/SQL has additional datatypes for constants and variables, which include `BOOLEAN`, reference types, composite types (collections and records), and user-defined subtypes.

See Also:

- *PL/SQL User's Guide and Reference* for information about PL/SQL datatypes and a summary of the characteristics of each Oracle datatype
- *Oracle9i Application Developer's Guide - Fundamentals* for information about how to use the built-in datatypes

The following sections that describe each of the built-in datatypes in more detail.

Character Datatypes

The character datatypes store character (alphanumeric) data in strings, with byte values corresponding to the character encoding scheme, generally called a character set or code page.

The database's character set is established when you create the database. Examples of character sets are 7-bit ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), Code Page 500, Japan Extended UNIX, and Unicode UTF-8. Oracle supports both single-byte and multibyte encoding schemes.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information about how to select a character datatype
- *Oracle9i Globalization Support Guide* for more information about converting character data

CHAR Datatype

The `CHAR` datatype stores fixed-length character strings. When you create a table with a `CHAR` column, you must specify a string length (in bytes or characters) between 1 and 2000 bytes for the `CHAR` column width. The default is 1 byte. Oracle then guarantees that:

- When you insert or update a row in the table, the value for the `CHAR` column has the fixed length.
- If you give a shorter value, then the value is blank-padded to the fixed length.
- If you give a longer value with trailing blanks, then blanks are trimmed from the value to the fixed length.

- If a value is too large, Oracle returns an error.

Oracle compares CHAR values using blank-padded comparison semantics.

> **See Also:** *Oracle9i SQL Reference* for details about blank-padded comparison semantics

VARCHAR2 and VARCHAR Datatypes

The VARCHAR2 datatype stores variable-length character strings. When you create a table with a VARCHAR2 column, you specify a maximum string length (in bytes or characters) between 1 and 4000 bytes for the VARCHAR2 column. For each row, Oracle stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle returns an error. Using VARCHAR2 and VARCHAR saves on space used by the table.

For example, assume you declare a column VARCHAR2 with a maximum size of 50 characters. In a single-byte character set, if only 10 characters are given for the VARCHAR2 column value in a particular row, the column in the row's row piece stores only the 10 characters (10 bytes), not 50.

Oracle compares VARCHAR2 values using nonpadded comparison semantics.

> **See Also:** *Oracle9i SQL Reference* for details about nonpadded comparison semantics

VARCHAR Datatype

The VARCHAR datatype is synonymous with the VARCHAR2 datatype. To avoid possible changes in behavior, always use the VARCHAR2 datatype to store variable-length character strings.

Length Semantics for Character Datatypes

The Oracle9*i* Globalization Support feature allows the use of various character sets for the character datatypes. Globalization Support enables you to process single-byte and multibyte character data and convert between character sets. Client sessions can use client character sets that are different from the database character set.

Consider the size of characters when you specify the column length for character datatypes. You must consider this issue when estimating space for tables with columns that contain character data.

The length semantics of character datatypes can be measured in bytes or characters.

- **Byte semantics** treat strings as a sequence of bytes. This is the default for character datatypes.
- **Character semantics** treat strings as a sequence of characters. A character is technically a codepoint of the database character set.

For single byte character sets, columns defined in character semantics are basically the same as those defined in byte semantics. Character semantics are useful for defining varying-width multibyte strings; it reduces the complexity when defining the actual length requirements for data storage. For example, in a Unicode database (UTF8), you need to define a VARCHAR2 column that can store up to five Chinese characters together with five English characters. In byte semantics, this would require (5*3 bytes) + (1*5 bytes) = 20 bytes; in character semantics, the column would require 10 characters.

VARCHAR2(20 BYTE) and SUBSTRB(<*string*>, 1, 20) use byte semantics. VARCHAR2(10 CHAR) and SUBSTR(<*string*>, 1, 10) use character semantics.

The parameter NLS_LENGTH_SEMANTICS decides whether a new column of character datatype uses byte or character semantics. The default length semantic is byte. If all character datatype columns in a database use byte semantics (or all use character semantics) then users do not have to worry about which columns use which semantics. The BYTE and CHAR qualifiers shown above should be avoided when possible, because they lead to mixed-semantics databases. Instead, the NLS_LENGTH_SEMANTICS initialization parameter should be set appropriately in INIT.ORA, and columns should use the default semantics.

See Also:

- "Use of Unicode Data in an Oracle Database" on page 13-7
- *Oracle9i Globalization Support Guide* for more information about Oracle's Globalization Support feature
- *Oracle9i Application Developer's Guide - Fundamentals* for information about setting length semantics and choosing the appropriate Unicode character set.
- *Oracle9i Database Migration* for information about migrating existing columns to character semantics

NCHAR and NVARCHAR2 Datatypes

NCHAR and NVARCHAR2 are Unicode data types that store Unicode character data. The character set of NCHAR and NVARCHAR2 datatypes can only be either AL16UTF16 or UTF8 and is specified at database creation time as the national character set. AL16UTF16 and UTF8 are both Unicode encoding.

- The NCHAR datatype stores fixed-length character strings that correspond to the national character set.
- The NVARCHAR2 datatype stores variable length character strings.

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

Example 13–1 Defining Maximum Byte Length of a Column

If national character set is UTF8, the following statement defines the maximum byte length of 90 bytes:

```
CREATE TABLE tab1 (col1 NCHAR(30));
```

This statement creates a column with maximum character length of 30. The maximum byte length is the multiple of the maximum character length and the maximum number of bytes in each character.

NCHAR

The maximum length of an NCHAR column is 2000 bytes. It can hold up to 2000 characters. The actual data is subject to the maximum byte limit of 2000. The two size constraints must be satisfied simultaneously at run time.

NVARCHAR2

The maximum length of an NVARCHAR2 column is 4000 bytes. It can hold up to 4000 characters. The actual data is subject to the maximum byte limit of 4000. The two size constraints must be satisfied simultaneously at run time.

See Also: *Oracle9i Globalization Support Guide* for more information about the NCHAR and NVARCHAR2 datatypes

Use of Unicode Data in an Oracle Database

Unicode is an effort to have a unified encoding of every character in every language known to man. It also provides a way to represent privately-defined characters. A database column that stores Unicode can store text written in any language.

Oracle users deploying globalized applications have a strong need to store Unicode data in Oracle databases. They need a datatype which is guaranteed to be Unicode regardless of the database character set.

Oracle9*i*, Release 1 (9.0.1), supports a reliable Unicode data type through NCHAR, NVARCHAR2, and NCLOB. These data types are guaranteed to be Unicode encoding and always use character length semantics. The character sets used by NCHAR/NVARCHAR2 can be either UTF8 or AL16UTF16, depending on the setting of the national character set when the database is created. These data types allow character data in Unicode to be stored in a database that may or may not use Unicode as database character set.

Implicit Type Conversion

To all the implicit conversions for CHAR/VARCHAR2, Oracle9*i*, Release 1 (9.0.1), adds support for implicit conversion for NCHAR/NVARCHAR2. Implicit conversion between CHAR/VARCHAR2 and NCHAR/NVARCHAR2 is also supported.

LOB Character Datatypes

The LOB datatypes for character data are CLOB and NCLOB. They can store up to 4 gigabytes of character data (CLOB) or national character set data (NCLOB). LOB datatypes are intended to replace the LONG datatype functionality.

See Also: "LOB Datatypes" on page 13-14

LONG Datatype

Note: The LONG datatype is provided for backward compatibility with existing applications. In new applications, use CLOB and NCLOB datatypes for large amounts of character data.

Columns defined as LONG can store variable-length character data containing up to 2 gigabytes of information. LONG data is text data that is to be appropriately converted when moving among different systems.

LONG datatype columns are used in the data dictionary to store the text of view definitions. You can use LONG columns in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information about the restrictions on the LONG datatype
- "RAW and LONG RAW Datatypes" on page 13-16 for information about the LONG RAW datatype

NUMBER Datatype

The NUMBER datatype stores fixed and floating-point numbers. Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems operating Oracle, up to 38 digits of precision.

The following numbers can be stored in a NUMBER column:

- Positive numbers in the range 1×10^{-130} to $9.99...9 \times 10^{125}$ with up to 38 significant digits
- Negative numbers from -1×10^{-130} to $9.99...99 \times 10^{125}$ with up to 38 significant digits
- Zero
- Positive and negative infinity (generated only by importing from an Oracle Version 5 database)

For numeric columns, you can specify the column as:

```
column_name NUMBER
```

Optionally, you can also specify a **precision** (total number of digits) and **scale** (number of digits to the right of the decimal point):

```
column_name NUMBER (precision, scale)
```

If a precision is not specified, the column stores values as given. If no scale is specified, the scale is zero.

Oracle guarantees portability of numbers with a precision equal to or less than 38 digits. You can specify a scale and no precision:

```
column_name NUMBER (*, scale)
```

In this case, the precision is 38, and the specified scale is maintained.

When you specify numeric fields, it is a good idea to specify the precision and scale. This provides extra integrity checking on input.

Table 13–1 shows examples of how data would be stored using different scale factors.

Table 13–1 How Scale Factors Affect Numeric Data Storage

Input Data	Specified As	Stored As
7,456,123.89	NUMBER	7456123.89
7,456,123.89	NUMBER(*,1)	7456123.9
7,456,123.89	NUMBER(9)	7456124
7,456,123.89	NUMBER(9,2)	7456123.89
7,456,123.89	NUMBER(9,1)	7456123.9
7,456,123.89	NUMBER(6)	(not accepted, exceeds precision)
7,456,123.89	NUMBER(7,-2)	7456100

If you specify a negative scale, Oracle rounds the actual data to the specified number of places to the left of the decimal point. For example, specifying (7,-2) means Oracle rounds to the nearest hundredths, as shown in Table 13–1.

For input and output of numbers, the standard Oracle default decimal character is a period, as in the number 1234.56. The decimal is the character that separates the integer and decimal parts of a number. You can change the default decimal character with the initialization parameter NLS_NUMERIC_CHARACTERS. You can also change it for the duration of a session with the ALTER SESSION statement. To enter numbers that do not use the current default decimal character, use the TO_NUMBER function.

Internal Numeric Format

Oracle stores numeric data in variable-length format. Each value is stored in scientific notation, with 1 byte used to store the exponent and up to 20 bytes to store the mantissa. The resulting value is limited to 38 digits of precision. Oracle does not

store leading and trailing zeros. For example, the number 412 is stored in a format similar to 4.12 x 10^2, with 1 byte used to store the exponent(2) and 2 bytes used to store the three significant digits of the mantissa(4,1,2). Negative numbers include the sign in their length.

Taking this into account, the column size in bytes for a particular numeric data value NUMBER(*p*), where *p* is the precision of a given value, can be calculated using the following formula:

```
ROUND((length(p)+s)/2))+1
```

where *s* equals zero if the number is positive, and *s* equals 1 if the number is negative.

Zero and positive and negative infinity (only generated on import from Version 5 Oracle databases) are stored using unique representations. Zero and negative infinity each require 1 byte; positive infinity requires 2 bytes.

DATE Datatype

The DATE datatype stores point-in-time values (dates and times) in a table. The DATE datatype stores the year (including the century), the month, the day, the hours, the minutes, and the seconds (after midnight).

Oracle can store dates in the Julian era, ranging from January 1, 4712 BCE through December 31, 4712 CE (Common Era). Unless BCE ('BC' in the format mask) is specifically used, CE date entries are the default.

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

For input and output of dates, the standard Oracle default date format is DD-MON-YY, as follows:

```
'13-NOV-92'
```

You can change this default date format for an instance with the parameter NLS_DATE_FORMAT. You can also change it during a user session with the ALTER SESSION statement. To enter dates that are not in standard Oracle date format, use the TO_DATE function with a format mask:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

> **Note:** If you use the standard date format DD-MON-YY, YY gives the year in the 20th century (for example, 31-DEC-92 is December 31, 1992). If you want to indicate years in any century other than the 20th century, use a different format mask, as shown previously.

Oracle stores time in 24-hour format—HH:MI:SS. By default, the time in a date field is 00:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as in:

```
INSERT INTO birthdays (bname, bday) VALUES
    ('ANDY',TO_DATE('13-AUG-66 12:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

Use of Julian Dates

Julian dates allow continuous dating by the number of days from a common reference. (The reference is 01-01-4712 years BCE, so current dates are somewhere in the 2.4 million range.) A Julian date is nominally a noninteger, the fractional part being a portion of a day. Oracle uses a simplified approach that results in integer values. Julian dates can be calculated and interpreted differently. The calculation method used by Oracle results in a seven-digit number (for dates most often used), such as 2449086 for 08-APR-93.

> **Note:** Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms.

The format mask 'J' can be used with date functions (TO_DATE or TO_CHAR) to convert date data into Julian dates. For example, the following query returns all dates in Julian date format:

```
SELECT TO_CHAR (hiredate, 'J') FROM emp;
```

You must use the TO_NUMBER function if you want to use Julian dates in calculations. You can use the TO_DATE function to enter Julian dates:

```
INSERT INTO emp (hiredate) VALUES (TO_DATE(2448921, 'J'));
```

Date Arithmetic

Oracle date arithmetic takes into account the anomalies of the calendars used throughout history. For example, the switch from the Julian to the Gregorian calendar, 15-10-1582, eliminated the previous 10 days (05-10-1582 through 14-10-1582). The year 0 does not exist.

You can enter missing dates into the database, but they are ignored in date arithmetic and treated as the next "real" date. For example, the next day after 04-10-1582 is 15-10-1582, and the day following 05-10-1582 is also 15-10-1582.

Note: This discussion of date arithmetic might not apply to all countries' date standards (such as those in Asia).

Centuries and the Year 2000

Oracle stores year data with the century information. For example, the Oracle database stores 1996 or 2001, and not simply 96 or 01. The `DATE` datatype always stores a four-digit year internally, and all other dates stored internally in the database have four digit years. Oracle utilities such as import, export, and recovery also deal with four-digit years.

Daylight Savings Support

Oracle9*i*, Release 1 (9.0.1), provides daylight savings support for `DATETIME` datatypes in the server. You can insert and query `DATETIME` values based on local time in a specific region. The `DATETIME` datatypes `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` are time-zone aware.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about centuries and date format masks
- *Oracle9i SQL Reference* for information about date format codes

Time Zones

Oracle9*i*, Release 1 (9.0.1), lets you include the time zone in your date/time data and provides support for fractional seconds. Three new datatypes are added to `DATE`, with the following differences:

Datatype	Time Zone	Fractional Seconds
`DATE`	No	No
`TIMESTAMP`	No	Yes
`TIMESTAMP WITH TIME ZONE`	Explicit	Yes
`TIMESTAMP WITH LOCAL TIME ZONE`	Relative	Yes

`TIMESTAMP WITH LOCAL TIME ZONE` is stored in the database time zone. When a user selects the data, the value is adjusted to the user's session time zone.

Example:

A San Francisco database has system time zone = -8:00. When a New York client (session time zone = -5:00) inserts into or selects from the San Francisco database, `TIMESTAMP WITH LOCAL TIME ZONE` data is adjusted as follows:

- The New York client inserts `TIMESTAMP'1998-1-23 6:00:00-5:00'` into a `TIMESTAMP WITH LOCAL TIME ZONE` column in the San Francisco database. The inserted data is stored in San Francisco as binary value `1998-1-23 3:00:00`.
- When the New York client selects that inserted data from the San Francisco database, the value displayed in New York is `'1998-1-23 6:00:00'`.
- A San Francisco client, selecting the same data, see the value `'1998-1-23 3:00:00'`.

Note: To avoid unexpected results in your DML operations on datatime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the database time zone or the session time zone has not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.

See Also: *Oracle9i SQL Reference* for details about the syntax of creating and entering data in time stamp columns

LOB Datatypes

The LOB datatypes BLOB, CLOB, NCLOB, and BFILE enable you to store large blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) up to 4 gigabytes in size. They provide efficient, random, piece-wise access to the data. Oracle Corporation recommends that you always use LOB datatypes over LONG datatypes.

You can perform parallel queries (but not parallel DML or DDL) on LOB columns.

LOB datatypes differ from LONG and LONG RAW datatypes in several ways. For example:

- A table can contain multiple LOB columns but only one LONG column.
- A table containing one or more LOB columns can be partitioned, but a table containing a LONG column cannot be partitioned.
- The maximum size of a LOB is 4 gigabytes, but the maximum size of a LONG is 2 gigabytes.
- LOBs support random access to data, but LONGs support only sequential access.
- LOB datatypes (except NCLOB) can be attributes of a user-defined object type but LONG datatypes cannot.
- Temporary LOBs that act like local variables can be used to perform transformations on LOB data. Temporary internal LOBs (BLOBs, CLOBs, and NCLOBs) are created in the user's temporary tablespace and are independent of tables. For LONG datatypes, however, no temporary structures are available.

- Tables with LOB columns can be replicated, but tables with LONG columns cannot.

SQL statements define LOB columns in a table and LOB attributes in a user-defined object type. When defining LOBs in a table, you can explicitly specify the tablespace and storage characteristics for each LOB.

LOB datatypes can be stored inline (within a table), out-of-line (within a tablespace, using a LOB locator), or in an external file (BFILE datatypes).

With compatibility set to Oracle9*i*, Release 1 (9.0.1), or higher, you can use LOBs with SQL VARCHAR operators and functions.

See Also:

- *Oracle9i SQL Reference* for a complete list of differences between the LOB datatypes and the LONG and LONG RAW datatypes
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*for more information about LOB storage and LOB locators

BLOB Datatype

The BLOB datatype stores unstructured binary data in the database. BLOBs can store up to 4 gigabytes of binary data.

BLOBs participate fully in transactions. Changes made to a BLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, BLOB locators cannot span transactions or sessions.

CLOB and NCLOB Datatypes

The CLOB and NCLOB datatypes store up to 4 gigabytes of character data in the database. CLOBs store database character set data and NCLOBs store Unicode national character set data. For varying-width database character sets, the CLOB value is stored in the database using the two-byte Unicode character set, which has a fixed width. Oracle translates the stored Unicode value to the character set requested on the client or on the server, which can be fixed-width or varying width. When you insert data into a CLOB column using a varying-width character set, Oracle converts the data into Unicode before storing it in the database.

CLOBs and NCLOBs participate fully in transactions. Changes made to a CLOB or NCLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, CLOB and NCLOB locators cannot span transactions or sessions.

You cannot create an object type with NCLOB attributes, but you can specify NCLOB parameters in a method for an object type.

> **See Also:** *Oracle9i Globalization Support Guide* for more information about national character set data and the Unicode character set

BFILE Datatype

The BFILE datatype stores unstructured binary data in operating-system files outside the database. A BFILE column or attribute stores a file locator that points to an external file containing the data. BFILEs can store up to 4 gigabytes of data.

BFILEs are read-only; you cannot modify them. They support only random (not sequential) reads, and they do not participate in transactions. The underlying operating system must maintain the file integrity, security, and durability for BFILEs. The database administrator must ensure that the file exists and that Oracle processes have operating-system read permissions on the file.

RAW and LONG RAW Datatypes

> **Note:** The LONG RAW datatype is provided for backward compatibility with existing applications. For new applications, use the BLOB and BFILE datatypes for large amounts of binary data.

The RAW and LONG RAW datatypes are used for data that is not to be interpreted (not converted when moving data between different systems) by Oracle. These datatypes are intended for binary data or byte strings. For example, LONG RAW can be used to store graphics, sound, documents, or arrays of binary data. The interpretation depends on the use.

RAW is a variable-length datatype like the VARCHAR2 character datatype, except Oracle Net Services (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Oracle Net Services and Import/Export automatically convert CHAR, VARCHAR2, and LONG data between the database character set and the user session character set (set by the NLS_LANGUAGE parameter of the ALTER SESSION statement), if the two character sets are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB.'

LONG RAW data cannot be indexed, but RAW data can be indexed.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information about other restrictions on the LONG RAW datatype

ROWID and UROWID Datatypes

Oracle uses a ROWID datatype to store the address *(***rowid***)* of every row in the database.

- **Physical rowids** store the addresses of rows in ordinary tables (excluding index-organized tables), clustered tables, table partitions and subpartitions, indexes, and index partitions and subpartitions.
- **Logical rowids** store the addresses of rows in index-organized tables.

A single datatype called the **universal rowid**, or UROWID, supports both logical and physical rowids, as well as rowids of foreign tables such as non-Oracle tables accessed through a gateway.

A column of the UROWID datatype can store all kinds of rowids. The value of the COMPATIBLE initialization parameter must be set to 8.1 or higher to use UROWID columns.

> **See Also:** "Rowids in Non-Oracle Databases" on page 13-24

The ROWID Pseudocolumn

Each table in an Oracle database internally has a **pseudocolumn** named ROWID. This pseudocolumn is not evident when listing the structure of a table by executing a SELECT * FROM ... statement, or a DESCRIBE ... statement using SQL*Plus, nor does the pseudocolumn take up space in the table. However, each row's address can be retrieved with a SQL query using the reserved word ROWID as a column name, for example:

```
SELECT ROWID, ename FROM emp;
```

You cannot set the value of the pseudocolumn ROWID in INSERT or UPDATE statements, and you cannot delete a ROWID value. Oracle uses the ROWID values in the pseudocolumn ROWID internally for the construction of indexes.

You can reference rowids in the pseudocolumn `ROWID` like other table columns (used in `SELECT` lists and `WHERE` clauses), but rowids are not stored in the database, nor are they database data. However, you can create tables that contain columns having the `ROWID` datatype, although Oracle does not guarantee that the values of such columns are valid rowids. The user must ensure that the data stored in the `ROWID` column truly is a valid `ROWID`.

See Also: "How Rowids Are Used" on page 13-21

Physical Rowids

Physical rowids provide the fastest possible access to a row of a given table. They contain the physical address of a row (down to the specific block) and allow you to retrieve the row in a single block access. Oracle guarantees that as long as the row exists, its rowid does not change. These performance and stability qualities make rowids useful for applications that select a set of rows, perform some operations on them, and then access some of the selected rows again, perhaps with the purpose of updating them.

Every row in a nonclustered table is assigned a unique rowid that corresponds to the physical address of a row's row piece (or the initial row piece if the row is chained among multiple row pieces). In the case of clustered tables, rows in different tables that are in the same data block can have the same rowid.

A row's assigned rowid remains unchanged unless the row is exported and imported using the Import and Export utilities. When you delete a row from a table and then commit the encompassing transaction, the deleted row's associated rowid can be assigned to a row inserted in a subsequent transaction.

A physical rowid datatype has one of two formats:

- The **extended rowid** format supports tablespace-relative data block addresses and efficiently identifies rows in partitioned tables and indexes as well as nonpartitioned tables and indexes. Tables and indexes created by an Oracle8*i* (or higher) server always have extended rowids.
- A **restricted rowid** format is also available for backward compatibility with applications developed with Oracle7 or earlier releases.

Extended Rowids

Extended rowids use a base 64 encoding of the physical address for each row selected. The encoding characters are `A-Z, a-z, 0-9, +,` and `/`. For example, the following query:

```
SELECT ROWID, ename FROM emp WHERE deptno = 20;
```

can return the following row information:

```
ROWID              ENAME
------------------ ----------
AAAAaoAATAAABrXAAA BORTINS
AAAAaoAATAAABrXAAE RUGGLES
AAAAaoAATAAABrXAAG CHEN
AAAAaoAATAAABrXAAN BLUMBERG
```

An extended rowid has a four-piece format, `OOOOOOFFFBBBBBBRRR:`

- `OOOOOO:` The **data object number** that identifies the database segment (`AAAAao` in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- `FFF:` The tablespace-relative **datafile number** of the datafile that contains the row (file `AAT` in the example).
- `BBBBBB:` The **data block** that contains the row (block `AAABrX` in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- `RRR:` The **row** in the block.

You can retrieve the data object number from data dictionary views `USER_OBJECTS`, `DBA_OBJECTS`, and `ALL_OBJECTS`. For example, the following query returns the data object number for the `EMP` table in the `SCOTT` schema:

```
SELECT DATA_OBJECT_ID FROM DBA_OBJECTS
    WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP';
```

You can also use the `DBMS_ROWID` package to extract information from an extended rowid or to convert a rowid from extended format to restricted format (or vice versa).

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information about the `DBMS_ROWID` package

Restricted Rowids

Restricted rowids use a binary representation of the physical address for each row selected. When queried using SQL*Plus, the binary representation is converted to a `VARCHAR2`/hexadecimal representation. The following query:

```
SELECT ROWID, ename FROM emp
    WHERE deptno = 30;
```

can return the following row information:

```
ROWID              ENAME
------------------ ----------
00000DD5.0000.0001 KRISHNAN
00000DD5.0001.0001 ARBUCKLE
00000DD5.0002.0001 NGUYEN
```

As shown, a restricted rowid's VARCHAR2/hexadecimal representation is in a three-piece format, **block.row.file**:

- The **data block** that contains the row (block DD5 in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- The **row** in the block that contains the row (rows 0, 1, 2 in the example). Row numbers of a given block always start with 0.
- The **datafile** that contains the row (file 1 in the example). The first datafile of every database is always 1, and file numbers are unique within a database.

Examples of Rowid Use

You can use the function SUBSTR to break the data in a rowid into its components. For example, you can use SUBSTR to break an extended rowid into its four components (database object, file, block, and row):

```
SELECT ROWID,
       SUBSTR(ROWID,1,6) "OBJECT",
       SUBSTR(ROWID,7,3) "FIL",
       SUBSTR(ROWID,10,6) "BLOCK",
       SUBSTR(ROWID,16,3) "ROW"
       FROM products;

ROWID              OBJECT  FIL  BLOCK   ROW
------------------ ------  ---  ------  ----
AAAA8mAALAAAAQkAAA AAAA8m  AAL  AAAAQk  AAA
AAAA8mAALAAAAQkAAF AAAA8m  AAL  AAAAQk  AAF
AAAA8mAALAAAAQkAAI AAAA8m  AAL  AAAAQk  AAI
```

Or you can use SUBSTR to break a restricted rowid into its three components (block, row, and file):

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FILE",
       SUBSTR(ROWID,1,8) "BLOCK",
       SUBSTR(ROWID,10,4) "ROW"
       FROM products;

ROWID              FILE  BLOCK     ROW
-----------------  ----  --------  ----
00000DD5.0000.0001  0001  00000DD5  0000
00000DD5.0001.0001  0001  00000DD5  0001
00000DD5.0002.0001  0001  00000DD5  0002
```

Rowids can be useful for revealing information about the physical storage of a table's data. For example, if you are interested in the physical location of a table's rows (such as for table striping), the following query of an extended rowid tells how many datafiles contain rows of a given table:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID,7,3))) "FILES" FROM tablename;

   FILES
--------
       2
```

See Also:

- *Oracle9i SQL Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i Database Performance Guide and Reference*

for more examples using rowids

How Rowids Are Used

Oracle uses rowids internally for the construction of indexes. Each key in an index 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 can be used to see how a table is organized.
- Rowids are unique identifiers for rows in a given table.

Before you use rowids in DML statements, they should be verified and guaranteed not to change. The intended rows should be locked so they cannot be deleted.

Under some circumstances, requesting data with an invalid rowid could cause a statement to fail.

You can also create tables with columns defined using the `ROWID` datatype. For example, you can define an exception table with a column of datatype `ROWID` to store the rowids of rows in the database that violate integrity constraints. Columns defined using the `ROWID` datatype behave like other table columns: values can be updated, and so on. Each value in a column defined as datatype `ROWID` requires six bytes to store pertinent column data.

Logical Rowids

Rows in index-organized tables do not have permanent physical addresses—they are stored in the index leaves and can move within the block or to a different block as a result of insertions. Therefore their row identifiers cannot be based on physical addresses. Instead, Oracle provides index-organized tables with logical row identifiers, called **logical rowids**, that are based on the table's primary key. Oracle uses these logical rowids for the construction of secondary indexes on index-organized tables.

Each logical rowid used in a secondary index can include a **physical guess**, which identifies the block location of the row in the index-organized table at the time the guess was made; that is, when the secondary index was created or rebuilt.

Oracle can use guesses to probe into the leaf block directly, bypassing the full key search. This ensures that rowid access of nonvolatile index-organized tables gives comparable performance to the physical rowid access of ordinary tables. In a volatile table, however, if the guess becomes stale the probe can fail, in which case a primary key search must be performed.

The values of two logical rowids are considered equal if they have the same primary key values but different guesses.

Comparison of Logical Rowids with Physical Rowids

Logical rowids are similar to the physical rowids in the following ways:

- Logical rowids are accessible through the `ROWID` pseudocolumn.

  You can use the `ROWID` pseudocolumn to select logical rowids from an index-organized table. The `SELECT ROWID` statement returns an opaque structure, which internally consists of the table's primary key and the physical guess (if any) for the row, along with some control information.

You can access a row using predicates of the form `WHERE ROWID = value`, where *value* is the opaque structure returned by `SELECT ROWID`.

- Access through the logical rowid is the fastest way to get to a specific row, although it can require more than one block access.
- A row's logical rowid does not change as long as the primary key value does not change. This is less stable than the physical rowid, which stays immutable through all updates to the row.
- Logical rowids can be stored in a column of the `UROWID` datatype

One difference between physical and logical rowids is that logical rowids cannot be used to see how a table is organized.

See Also: "ROWID and UROWID Datatypes" on page 13-17

Guesses in Logical Rowids

When a row's physical location changes, the logical rowid remains valid even if it contains a guess, although the guess could become stale and slow down access to the row. Guess information cannot be updated dynamically. For secondary indexes on index-organized tables, however, you can rebuild the index to obtain fresh guesses. Note that rebuilding a secondary index on an index-organized table involves reading the base table, unlike rebuilding an index on an ordinary table.

Collect index statistics with the `DBMS_STATS` package or `ANALYZE` statement to keep track of the staleness of guesses, so Oracle does not use them unnecessarily. This is particularly important for applications that store rowids with guesses persistently in a `UROWID` column, then retrieve the rowids later and use them to fetch rows.

When you collect index statistics with the `DBMS_STATS` package or `ANALYZE` statement, Oracle checks whether the existing guesses are still valid and records the percentage of stale / valid guesses in the data dictionary. After you rebuild a secondary index (recomputing the guesses), collect index statistics again.

In general, logical rowids without guesses provide the fastest possible access for a highly volatile table. If a table is static or if the time between getting a rowid and using it is sufficiently short to make row movement unlikely, logical rowids with guesses provide the fastest access.

See Also: *Oracle9i Database Performance Guide and Reference* for more information about collecting statistics

Rowids in Non-Oracle Databases

Oracle database applications can be executed against non-Oracle database servers using SQL*Connect or the Oracle Transparent Gateway. In such cases, the format of rowids varies according to the characteristics of the non-Oracle system. Furthermore, no standard translation to VARCHAR2/hexadecimal format is available. Programs can still use the ROWID datatype. However, they must use a nonstandard translation to hexadecimal format of length up to 256 bytes.

Rowids of a non-Oracle database can be stored in a column of the UROWID datatype.

See Also:

- *Oracle Call Interface Programmer's Guide* for further details on handling rowids with non-Oracle systems
- "ROWID and UROWID Datatypes" on page 13-17

ANSI, DB2, and SQL/DS Datatypes

The ANSI datatype conversions to Oracle datatypes are shown in Table 13–2. The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, *s* (scale) defaults to 0.

Table 13–2 ANSI Datatype Conversions to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER(*n*), CHAR(*n*)	CHAR (*n*)
NUMERIC(*p,s*), DECIMAL(*p,s*), DEC(*p,s*)	NUMBER (*p,s*)
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT(*p*)	FLOAT (*p*)
REAL	FLOAT (63)
DOUBLE PRECISION	FLOAT(126)
CHARACTER VARYING(*n*), CHAR VARYING(*n*)	VARCHAR2(*n*)

The IBM products SQL/DS, and DB2 datatypes TIME, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC have no corresponding Oracle datatype and cannot be used. The TIME datatype is a subcomponent of the Oracle datatype DATE.

Table 13–3 shows the DB2 and SQL/DS conversions.

Table 13–3 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

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

XML Datatypes

Oracle9*i*, Release 1 (9.0.1), provides two new datatypes to handle XML data.

- XMLType allows storage of XML documents in NCLOB or structured object types.
- XMLIndex, installed under user SYSXML, automatically indexes the XML schema associated with each XML document.

XMLType

XMLType is an opaque type which supports storage of XML documents in NCLOB or structured object types.

XMLType with NCLOB Storage

To use XMLType with NCLOB storage, you define a column of XML type. You can then insert any XML document in this column. The XML data is stored in a NCLOB; no extra columns are needed.

This XMLType without any indexes can store any XML document. Because the XMLType is a LOB column, it can be imported, exported, and loaded in the same way as regular LOB columns.

XMLType with Structured Storage

To use XMLType with structured storage, you specify the following when you define a column of XMLType:

- The XML schema to use
- The root element of the schema to map
- The template to use to perform the mapping

The template specifies the mapping between the XML schema and the object type. The template can be either created by the user or generated by default by the system. This template is registered in the template registry.

The `XML` type itself is made into a virtual column and a hidden object type column is created to store the object type instances. A dependency is created on the template and the schema.

When you insert an `XMLType` instance into this column, an object instance is created from the XML document, using the template. This object instance is inserted into the hidden column. You can also choose to keep the `NCLOB` as real instead of virtual. This is useful where the whole document needs to be kept intact for auditing or verification later. In this case, the data is written to the `LOB` and broken into the object instance.

A column defined as an `XMLType` mapped to an ADT, can contain only XML documents which conform to that particular XML schema.

XMLIndex

In Oracle9*i*, Release 1 (9.0.1), an `XMLIndex` index type is installed under user `SYSXML.` `XMLIndex` automatically indexes the XML schema associated with each XML document, provided the documents conform to existing registered XML schemas.

You can also use `XMLIndex` to constrain the list of schemas to which the documents are allowed conform. You specify this in the parameters clause when you create the index type.

In addition to `XMLIndex,` you can also create Intermedia Text indexes on the XML data. This lets you use the `CONTAINS` operator for more complex searches.

See Also:

- *Oracle9i Application Developer's Guide - XML*
- *Oracle9i Application Developer's Guide - Advanced Queuing* for information about using `XMLType` with Oracle Advanced Queuing

Data Conversion

In some cases, Oracle supplies data of one datatype where it expects data of a different datatype. This is allowed when Oracle can automatically convert the data to the expected datatype. These are some of the functions used:

```
TO_NUMBER()
TO_CHAR()
TO_NCHAR()
TO_DATE()
TO_CLOB()
TO_NCLOB()
CHARTOROWID()
ROWIDTOCHAR()
ROWIDTONCHAR()
HEXTORAW()
RAWTOHEX()
RAWTONHEX()
REFTOHEX()
```

See Also: *Oracle9i SQL Reference* for the rules for implicit datatype conversions

14

User-Defined Datatypes

Object types and other user-defined datatypes allow you to define datatypes that model the structure and behavior of the data in their applications.

This chapter contains the following major sections:

- Introduction to User-Defined Datatypes
- User-Defined Datatype Categories
- Type Inheritance
- User-Defined Aggregate Functions
- Application Interfaces
- Datatype Evolution

Introduction to User-Defined Datatypes

Relational database management systems (RDBMSs) are the standard tool for managing business data. They provide reliable access to huge amounts of data for millions of businesses around the world every day.

Oracle is an **object-relational** database management system (ORDBMS), which means that users can define additional kinds of data—specifying both the structure of the data and the ways of operating on it—and use these types within the relational model. This approach adds value to the data stored in a database. User-defined datatypes make it easier for application developers to work with complex data such as images, audio, and video. Object types store structured business data in its natural form and allow applications to retrieve it that way. For that reason, they work efficiently with applications developed using object-oriented programming techniques.

Complex Data Models

The Oracle server allows you to define complex business models in SQL and make them part of your database schema. Applications that manage and share your data need only contain the application logic, not the data logic.

Complex Data Model Example

For example, your firm might use purchase orders to organize its purchasing, accounts payable, shipping, and accounts receivable functions.

A purchase order contains an associated supplier or customer and an indefinite number of line items. In addition, applications often need dynamically computed status information about purchase orders. For example, you may need the current value of the shipped or unshipped line items.

Later sections of this chapter show how you can define a schema object, called an **object type**, that serves as a template for all purchase order data in your applications. An object type specifies the elements, called **attributes**, that make up a structured data unit, such as a purchase order. Some attributes, such as the list of line items, can be other structured data units. The object type also specifies the operations, called **methods**, you can perform on the data unit, such as determining the total value of a purchase order.

You can create purchase orders that match the template and store them in table columns, just as you would numbers or dates.

You can also store purchase orders in **object tables**, where each row of the table corresponds to a single purchase order and the table columns are the purchase order's attributes.

Because the logic of the purchase order's structure and behavior is in your schema, your applications do not need to know the details and do not have to keep up with most changes.

Oracle uses schema information about object types to achieve substantial transmission efficiencies. A client-side application can request a purchase order from the server and receive all the relevant data in a single transmission. The application can then, without knowing storage locations or implementation details, navigate among related data items without further transmissions from the server.

Multimedia Datatypes

Many efficiencies of database systems arise from their optimized management of basic datatypes like numbers, dates, and characters. Facilities exist for comparing values, determining their distributions, building efficient indexes, and performing other optimizations.

Text, video, sound, graphics, and spatial data are examples of important business entities that do not fit neatly into those basic types. Oracle Enterprise Edition supports modeling and implementation of these complex datatypes.

User-Defined Datatype Categories

There are two categories of user-defined datatypes:

- Object types
- Collection types

User-defined datatypes use the built-in datatypes and other user-defined datatypes as the building blocks for datatypes that model the structure and behavior of data in applications.

User-defined types are schema objects. Their use is subject to the same kinds of administrative control as other schema objects.

See Also:

- Chapter 13, "System-Provided Datatypes"
- *Oracle9i Application Developer's Guide - Object-Relational Features*

Object Types

Object types are abstractions of the real-world entities—for example, purchase orders—that application programs deal with. An object type is a schema object with three kinds of components:

- A **name**, which serves to identify the object type uniquely within that schema
- **Attributes**, which model the structure and state of the real-world entity. Attributes are built-in types or other user-defined types.
- **Methods**, which are functions or procedures written in PL/SQL or Java and stored in the database, or written in a language such as C and stored externally. Methods implement operations the application can perform on the real-world entity.

An object type is a template. A structured data unit that matches the template is called an **object**.

Purchase Order Example

Here is an example of how you can define object types called `EXTERNAL_PERSON`, `LINEITEM`, and `PURCHASE_ORDER`.

The object types `EXTERNAL_PERSON` and `LINEITEM` have attributes of built-in types. The object type `PURCHASE_ORDER` has a more complex structure, which closely matches the structure of real purchase orders.

The attributes of `PURCHASE_ORDER` are `ID`, `CONTACT`, and `LINEITEMS`. The attribute `CONTACT` is an object, and the attribute `LINEITEMS` is a nested table.

```
CREATE TYPE external_person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TYPE lineitem AS OBJECT (
  item_name   VARCHAR2(30),
  quantity    NUMBER,
  unit_price  NUMBER(12,2) );

CREATE TYPE lineitem_table AS TABLE OF lineitem;

CREATE TYPE purchase_order AS OBJECT (
  id          NUMBER,
  contact     external_person,
  lineitems   lineitem_table,
```

```
  MEMBER FUNCTION
  get_value   RETURN NUMBER );
```

This is a simplified example. It does not show how to specify the body of the method `GET_VALUE`, nor does it show the full complexity of a real purchase order.

An object type is a template. Defining it does not result in storage allocation. You can use `LINEITEM`, `EXTERNAL_PERSON`, or `PURCHASE_ORDER` in SQL statements in most of the same places you can use types like `NUMBER` or `VARCHAR2`.

For example, you can define a relational table to keep track of your contacts:

```
CREATE TABLE contacts (
  contact     external_person
  date        DATE );
```

The `CONTACT` table is a relational table with an object type defining one of its columns. Objects that occupy columns of relational tables are called **column objects**.

See Also:

- "Nested Tables Description" on page 14-12
- "Row Objects and Column Objects" on page 14-8
- *Oracle9i Application Developer's Guide - Object-Relational Features* for a complete purchase order example

Types of Methods

Methods of an object type model the behavior of objects. The methods of an object type broadly fall into these categories:

- A **Member** method is a function or a procedure that always has an implicit `SELF` parameter as its first parameter, whose type is the containing object type.
- A **Static** method is a function or a procedure that does not have an implicit `SELF` parameter. Such methods can be invoked by qualifying the method with the type name, as in `TYPE_NAME.METHOD()`. Static methods are useful for specifying user-defined constructors or cast methods.
- **Comparison** methods are used for comparing instances of objects.

Oracle supports the choice of implementing type methods in PL/SQL, JAVA, and C.

In the example, PURCHASE_ORDER has a method named GET_VALUE. Each purchase order object has its own GET_VALUE method. For example, if *x* and *y* are PL/SQL variables that hold purchase order objects and *w* and *z* are variables that hold numbers, the following two statements can leave *w* and *z* with different values:

```
w = x.get_value();
z = y.get_value();
```

After those statements, *w* has the value of the purchase order referred to by variable *x*; *z* has the value of the purchase order referred to by variable *y*.

The term *x*.GET_VALUE () is an invocation of the method GET_VALUE. Method definitions can include parameters, but GET_VALUE does not need them, because it finds all of its arguments among the attributes of the object to which its invocation is tied. That is, in the first of the sample statements, it computes its value using the attributes of purchase order *x*. In the second it computes its value using the attributes of purchase order *y*. This is called the **selfish style** of method invocation.

Every object type also has one implicitly defined method that is not tied to specific objects, the object type's constructor method.

Object Type Constructor Methods Every object type has a system-defined **constructor method**; that is, a method that makes a new object according to the object type's specification. The name of the constructor method is the name of the object type. Its parameters have the names and types of the object type's attributes. The constructor method is a function. It returns the new object as its value.

For example, the expression:

```
purchase_order(
  1000376,
  external_person ("John Smith","1-800-555-1212"),
  NULL )
```

represents a purchase order object with the following attributes:

```
id         1000376
contact    external_person("John Smith","1-800-555-1212")
lineitems  NULL
```

The expression external_person ("John Smith", "1-800-555-1212") is an invocation of the constructor function for the object type EXTERNAL_PERSON. The object that it returns becomes the contact attribute of the purchase order.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for a discussion of null objects and null attributes

Comparison Methods Methods play a role in comparing objects. Oracle has facilities for comparing two data items of a given built-in type (for example, two numbers), and determining whether one is greater than, equal to, or less than the other. Oracle cannot, however, compare two items of an arbitrary user-defined type without further guidance from the definer. Oracle provides two ways to define an order relationship among objects of a given object type: map methods and order methods.

Map methods use Oracle's ability to compare built-in types. Suppose, for example, that you have defined an object type called `RECTANGLE`, with attributes `HEIGHT` and `WIDTH`. You can define a map method area that returns a number, namely the product of the rectangle's `HEIGHT` and `WIDTH` attributes. Oracle can then compare two rectangles by comparing their areas.

Order methods are more general. An order method uses its own internal logic to compare two objects of a given object type. It returns a value that encodes the order relationship. For example, it could return -1 if the first is smaller, 0 if they are equal, and 1 if the first is larger.

Suppose, for example, that you have defined an object type called `ADDRESS`, with attributes `STREET`, `CITY`, `STATE`, and `ZIP`. **Greater than** and **less than** may have no meaning for addresses in your application, but you may need to perform complex computations to determine when two addresses are equal.

In defining an object type, you can specify either a map method or an order method for it, but not both. If an object type has no comparison method, Oracle cannot determine a greater than or less than relationship between two objects of that type. It can, however, attempt to determine whether two objects of the type are equal.

Oracle compares two objects of a type that lacks a comparison method by comparing corresponding attributes:

- If all the attributes are non-null and equal, Oracle reports that the objects are equal.
- If there is an attribute for which the two objects have unequal non-null values, Oracle reports them unequal.
- Otherwise, Oracle reports that the comparison is not available (null).

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for examples of how to specify and use comparison methods

Object Tables

An **object table** is a special kind of table that holds objects and provides a relational view of the attributes of those objects.

For example, the following statement defines an object table for objects of the EXTERNAL_PERSON type defined earlier:

```
CREATE TABLE external_person_table OF external_person;
```

Oracle allows you to view this table in two ways:

- A single column table in which each entry is an EXTERNAL_PERSON object.
- A multicolumn table in which each of the attributes of the object type EXTERNAL_PERSON, namely NAME and PHONE, occupies a column

For example, you can execute the following instructions:

```
INSERT INTO external_person_table VALUES (
       "John Smith",
       "1-800-555-1212" );

SELECT VALUE(p) FROM external_person_table p
        WHERE p.name = "John Smith";
```

The first instruction inserts an EXTERNAL_PERSON object into EXTERNAL_PERSON_TABLE as a multicolumn table. The second selects from EXTERNAL_PERSON_TABLE as a single column table.

Row Objects and Column Objects Objects that appear in object tables are called **row objects**. Objects that appear in table columns or as attributes of other objects are called **column objects**.

Object Identifiers

Every row object in an object table has an associated logical object identifier (OID). Oracle assigns a unique system-generated identifier of length 16 bytes as the OID for each row object by default.

The OID column of an object table is a hidden column. Although the OID value in itself is not very meaningful to an object-relational application, Oracle uses this value to construct object references to the row objects. Applications need to be concerned with only object references that are used for fetching and navigating objects.

The purpose of the OID for a row object is to uniquely identify it in an object table. To do this Oracle implicitly creates and maintains an index on the OID column of an object table. The system-generated unique identifier has many advantages, among which are the unambiguous identification of objects in a distributed and replicated environment.

Primary-Key Based Object Identifiers For applications that do not require the functionality provided by globally unique system-generated identifiers, storing 16 extra bytes with each object and maintaining an index on it may not be efficient. Oracle allows the option of specifying the primary key value of a row object as the object identifier for the row object.

Primary-key based identifiers also have the advantage of enabling a more efficient and easier loading of the object table. By contrast, system-generated object identifiers need to be remapped using some user-specified keys, especially when references to them are also stored persistently.

Object Views Description

An object view is a virtual object table. Its rows are row objects. Oracle materializes object identifiers, which it does not store persistently, from primary keys in the underlying table or view.

> **See Also:** Chapter 15, "Object Views"

REFs

In the relational model, foreign keys express many-to-one relationships. Oracle object types provide a more efficient means of expressing many-to-one relationships when the "one" side of the relationship is a row object.

Oracle provides a built-in datatype called `REF` to encapsulate references to row objects of a specified object type. From a modeling perspective, `REFs` provide the ability to capture an association between two row objects. Oracle uses object identifiers to construct such `REF`s.

You can use a `REF` to examine or update the object it refers to. You can also use a `REF` to obtain a copy of the object it refers to. The only changes you can make to a `REF` are to replace its contents with a reference to a different object of the same object type or to assign it a null value.

Scoped REFs In declaring a column type, collection element, or object type attribute to be a `REF`, you can constrain it to contain only references to a specified object

table. Such a REF is called a **scoped** REF. Scoped REFs require less storage space and allow more efficient access than unscoped REFs.

Dangling REFs It is possible for the object identified by a REF to become unavailable through either deletion of the object or a change in privileges. Such a REF is called **dangling**. Oracle SQL provides a predicate (called IS DANGLING) to allow testing REFs for this condition.

Dereference REFs Accessing the object referred to by a REF is called **dereferencing** the REF. Oracle provides the DEREF operator to do this. Dereferencing a dangling REF results in a null object.

Oracle provides **implicit dereferencing** of REFs. For example, consider the following:

```
CREATE TYPE person AS OBJECT (
  name    VARCHAR2(30),
  manager REF person );
```

If *x* represents an object of type PERSON, then the expression:

```
x.manager.name
```

represents a string containing the NAME attribute of the PERSON object referred to by the MANAGER attribute of *x*. The previous expression is a shortened form of:

```
y.name, where y = DEREF(x.manager)
```

Obtain REFs You can obtain a REF to a row object by selecting the object from its object table and applying the REF operator. For example, you can obtain a REF to the purchase order with identification number 1000376 as follows:

```
DECLARE OrderRef REF to purchase_order;

SELECT REF(po) INTO OrderRef
               FROM purchase_order_table po
               WHERE po.id = 1000376;
```

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for examples of how to use REFs

Collection Types

Each collection type describes a data unit made up of an indefinite number of elements, all of the same datatype. The collection types are **array types** and **table types**.

Array types and table types are schema objects. The corresponding data units are called **VARRAYs** and **nested tables**. When there is no danger of confusion, we often refer to the collection types as VARRAYs and nested tables.

Collection types have constructor methods. The name of the constructor method is the name of the type, and its argument is a comma-separated list of the new collection's elements. The constructor method is a function. It returns the new collection as its value.

An expression consisting of the type name followed by empty parentheses represents a call to the constructor method to create an empty collection of that type. An empty collection is different from a null collection.

VARRAYs

An **array** is an ordered set of data **elements**. All elements of a given array are of the same datatype. Each element has an **index**, which is a number corresponding to the element's position in the array.

The number of elements in an array is the **size** of the array. Oracle allows arrays to be of variable size, which is why they are called VARRAYs. You must specify a maximum size when you declare the array type.

For example, the following statement declares an array type:

```
CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);
```

The VARRAYs of type PRICES have no more than 10 elements, each of datatype NUMBER(12,2).

Creating an array type does not allocate space. It defines a datatype, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type.

A VARRAY is normally stored in line; that is, in the same tablespace as the other data in its row. If it is sufficiently large, however, Oracle stores it as a BLOB.

See Also: *Oracle9i Application Developer's Guide - Object-Relational Features* for more information about using VARRAYs

Nested Tables Description

A **nested table** is an unordered set of data **elements**, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If an object type, the table can also be viewed as a multicolumn table, with a column for each attribute of the object type. If compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher, nested tables can contain other nested tables.

For example, in the purchase order example, the following statement declares the table type used for the nested tables of line items:

```
CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

A table type definition does not allocate space. It defines a type, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

When a table type appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table. For example, the following statement defines an object table for the object type PURCHASE_ORDER:

```
CREATE TABLE purchase_order_table OF purchase_order
   NESTED TABLE lineitems STORE AS lineitems_table;
```

The second line specifies LINEITEMS_TABLE as the storage table for the LINEITEMS attributes of all of the PURCHASE_ORDER objects in PURCHASE_ORDER_TABLE.

A convenient way to access the elements of a nested table individually is to use a nested cursor.

See Also:

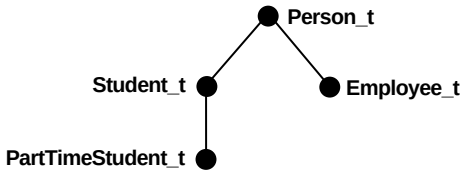
- *Oracle9i Database Reference* for information about nested cursors
- *Oracle9i Application Developer's Guide - Object-Relational Features* for more information about using nested tables

Type Inheritance

An object type can be created as a subtype of an existing object type. A single inheritance model is supported: the subtype can be derived from only one parent type. A type inherits all the attributes and methods of its direct supertype. It can add new attributes and methods, and it can override any of the inherited methods.

Figure 14–1 illustrates two subtypes, `Student_t` and `Employee_t`, created under `Person_t`.

Figure 14–1 A Type Hierarchy



Furthermore, a subtype can itself be refined by defining another subtype under it, thus building up type hierarchies. In the diagram above, `PartTimeStudent_t` is derived from subtype `Student_t`.

FINAL and NOT FINAL Types

A type declaration must have the `NOT FINAL` keyword, if you want it to have subtypes. The default is that the type is `FINAL;` that is, no subtypes can be created for the type. This allows for backward compatibility.

Example of Creating a NOT FINAL Object Type

```
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

`Person_t` is declared to be a `NOT FINAL` type. This enables definition of subtypes of `Person_t`.

`FINAL` types can be altered to be `NOT FINAL`. In addition, `NOT FINAL` types with no subtypes can be altered to be `FINAL`.

NOT INSTANTIABLE Types and Methods

A type can be declared to be NOT INSTANTIABLE. This implies that there is no constructor (default or user-defined) for the type. Thus, it will not be possible to construct instances of this type. The typical usage would be define instantiable subtypes for such a type, as follows:

```
CREATE TYPE Address_t AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_t UNDER Address_t(...);
CREATE TYPE IntlAddress_t UNDER Address_t(...);
```

A method of a type can be declared to be NOT INSTANTIABLE. Declaring a method as NOT INSTANTIABLE means that the type is not providing an implementation for that method. Furthermore, a type that contains any non-instantiable methods must necessarily be declared NOT INSTANTIABLE.

For example:

```
CREATE TYPE T AS OBJECT
(
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE;
```

A subtype of a NOT INSTANTIABLE type can override any of the non-instantiable methods of the supertype and provide concrete implementations. If there are any non-instantiable methods remaining, the subtype must also necessarily be declared NOT INSTANTIABLE.

A non-instantiable subtype can be defined under an instantiable supertype. Declaring a non-instantiable type to be FINAL is not allowed.

See Also: *PL/SQL User's Guide and Reference*

User-Defined Aggregate Functions

Oracle8*i* supports a fixed set of aggregate functions such as MAX, MIN, and SUM. Oracle9*i*, Release 1 (9.0.1), adds to this a mechanism for users to implement new aggregate functions with user-defined aggregation logic.

Why Have User-Defined Aggregate Functions?

User-Defined Aggregate functions (UDAGs) refer to aggregate functions with user-specified aggregation semantics. Users can create a new aggregate function and provide the aggregation logic through a set of routines. Once created, the

user-defined aggregate function can be used in SQL DML statements in a manner similar to built-in aggregates. The Oracle server evaluates the UDAG by invoking the user-provided aggregation routines appropriately.

Databases are increasingly being used to store complex data such as image, spatial, audio, video, etc. The complex data is typically stored in the database using object types, opaque types and / or LOBs. User-defined aggregates are primarily useful in specifying aggregation over such new domains of data.

Furthermore, UDAGs can be used to create new aggregate functions over traditional scalar data types for financial or scientific applications. Since, it is not possible to provide native support for all forms of aggregates, it is desirable to provide application developers with a flexible mechanism to add new aggregate functions.

See Also:

- *Oracle9i Data Cartridge Developer's Guide* for information about implementing user-defined aggregates
- *Oracle9i Data Warehousing Guide* for more information about using UDAGs in data warehousing

Creation and Use of UDAGs

The following is the procedure for implementing user-defined aggregates:

1. Implement the `ODCIAggregate` interface routines as methods of an object type.
2. Create a UDAG, using the `CREATE FUNCTION` command and specify the implementation type created in Step 1:

   ```
   CREATE FUNCTION MyUDAG ... AGGREGATE USING MyUDAGRoutines;
   ```

3. Use the UDAG in SQL DML statements the same way you use built-in aggregates:

   ```
   SELECT col1, MyUDAG(col2) FROM tab GROUP BY col1;
   ```

How Do Aggregate Functions Work?

An aggregate function conceptually takes a set of values as input and returns a single value. The sets of values for aggregation are typically identified using a `GROUP BY` clause. For example:

```
SELECT AVG(T.Sales)
FROM AnnualSales T
GROUP BY T.State
```

The evaluation of an aggregate function can be decomposed into three primitive operations. Considering the above example of `AVG()`, they are:

1. Initialize : initialize the computation

   ```
   runningSum = 0; runningCount = 0;
   ```

2. Iterate : process new input value

   ```
   runningSum += inputval; runningCount++;
   ```

3. Terminate : compute the result

   ```
   return (runningSum/runningCount);
   ```

The variables `runningSum` and `runningCount`, in the above example, determine the **state** of the aggregation. Thus, the **aggregation context** can be viewed as an object that contains `runningSum` and `runningCount` attributes. The Initialize method initializes the aggregation context, Iterate updates it and Terminate method uses the context to return the resultant aggregate value.

In addition, we require one more primitive operation to merge two aggregation contexts and create a new context. This operation is needed to combine the results of aggregation over subsets and obtain the aggregate over the entire set. This situation can arise during both serial and parallel evaluations of the aggregate.

4. Merge: combine the two aggregation contexts and return a single context

   ```
   runningSum = runningSum1 + runningSum2;
   runningCount = runningCount1 + runningCount2;
   ```

Oracle9*i*, Release 1 (9.0.1), allows the user to register new aggregate functions by providing specific (new) implementations for the above primitive operations.

Application Interfaces

Oracle provides several facilities for using user-defined datatypes in application programs:

SQL	PL/SQL	Pro*C/C++	OCI
OTT	JPublisher	JDBC	SQLJ

SQL

Oracle SQL data definition language provides the following support for user-defined datatypes:

- Defining object types, nested tables, and arrays
- Specifying privileges
- Specifying table columns of user-defined types
- Creating object tables

Oracle SQL data manipulation language provides the following support for user-defined datatypes:

- Querying and updating objects and collections
- Manipulating `REF`s

> **See Also:** *Oracle9i SQL Reference* for a complete description of SQL syntax

PL/SQL

PL/SQL is a procedural language that extends SQL. It offers features such as packages, data encapsulation, information hiding, overloading, and exception handling. Most stored procedures are written in PL/SQL.

PL/SQL allows use from within functions and procedures of the SQL features that support user-defined types. The parameters and variables of PL/SQL functions and procedures can be of user-defined types.

PL/SQL provides all the capabilities necessary to implement the methods associated with object types. These methods (functions and procedures) reside on the server as part of a user's schema.

> **See Also:** *PL/SQL User's Guide and Reference* for a complete description of PL/SQL

Pro*C/C++

The Oracle Pro*C/C++ precompiler allows programmers to use user-defined datatypes in C and C++ programs. Pro*C developers can use the Object Type Translator to map Oracle object types and collections into C datatypes to be used in the Pro*C application.

Pro*C provides compile time type checking of object types and collections and automatic type conversion from database types to C datatypes. Pro*C includes an EXEC SQL syntax to create and destroy objects and offers two ways to access objects in the server:

- SQL statements and PL/SQL functions or procedures embedded in Pro*C programs
- A simple interface to the object cache, where objects can be accessed by traversing pointers, then modified and updated on the server

See Also:

- "OCI" on page 14-19
- *Pro*C/C++ Precompiler Programmer's Guide* for a complete description of the Pro*C precompiler

Dynamic Creation and Access of Type Descriptions

Oracle9*i*, Release 1 (9.0.1), provide a C API to enable dynamic creation and access of type descriptions. Additionally, you can create transient type descriptions, type descriptions that are not stored persistently in the DBMS.

The C API enables creation and access of `OCIAnyData` and `OCIAnyDataSet.`

- The `OCIAnyData` type models a self descriptive (with regard to type) data instance of a given type.
- The `OCIAnyDataSet` type models a set of data instances of a given type.

Oracle9*i*, Release 1 (9.0.1), also provides SQL data types (in Oracle's Open Type System) that correspond to these data types.

- `SYS.ANYTYPE` corresponds to `OCIType`
- `SYS.ANYDATA` corresponds to `OCIAnyData`
- `SYS.ANYDATASET` corresponds to `OCIAnyDataSet`

You can create database table columns and SQL queries on such data.

The new C API uses the following terms:

- **Transient types** - Type descriptions (type metadata) that are not stored persistently in the Database.
- **Persistent types** - SQL types created using the `CREATE TYPE SQL` statement. Their type descriptions are stored persistently in the Database.

- **Self-descriptive data** - Data encapsulating type information along with the actual contents. The `ANYDATA` type `(OCIAnyData)` models such data. A data value of any SQL type can be converted to an `ANYDATA`, which can be converted back to the old data value. An incorrect conversion attempt results in an exception.
- **Self-descriptive MultiSet** - Encapsulation of a set of data instances (all of the same type), along with their type description.

See Also:

- *Oracle9i Application Developer's Guide - Object-Relational Features*
- *Oracle Call Interface Programmer's Guide*

OCI

The Oracle call interface (OCI) is a set of C language interfaces to the Oracle server. It provides programmers great flexibility in using the server's capabilities.

An important component of OCI is a set of calls to allow application programs to use a workspace called the object cache. The **object cache** is a memory block on the client side that allows programs to store entire objects and to navigate among them without round trips to the server.

The object cache is completely under the control and management of the application programs using it. The Oracle server has no access to it. The application programs using it must maintain data coherency with the server and protect the workspace against simultaneous conflicting access.

OCI provides functions to:

- Access objects on the server using SQL
- Access, manipulate and manage objects in the object cache by traversing pointers or `REF`s
- Convert Oracle dates, strings and numbers to C data types
- Manage the size of the object cache's memory
- Create transient type descriptions. Transient type descriptions are not stored persistently in the DBMS. Compatibility must be set to Oracle9*i*, Release 1 (9.0.1) or higher.

OCI improves concurrency by allowing individual objects to be locked. It improves performance by supporting complex object retrieval.

OCI developers can use the object type translator to generate the C datatypes corresponding to a Oracle object types.

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

OTT

The Oracle type translator (OTT) is a program that automatically generates C language structure declarations corresponding to object types. OTT facilitates using the Pro*C precompiler and the OCI server access package.

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*

JPublisher

Java Publisher (JPublisher) is a program that automatically generates Java class definitions corresponding to user-defined types in the database. Java Publisher facilitates using SQLJ and the JDBC server access package.

See Also: *Oracle9i JPublisher User's Guide*

JDBC

Java Database Connectivity (JDBC) is a set of Java interfaces to the Oracle server. Oracle's JDBC:

- Allows access to objects and collection types defined in the database from Java programs through dynamic SQL
- Provides for translation of types defined in the database into Java classes through default or customizable mappings

See Also: *Oracle9i JDBC Developer's Guide and Reference*

SQLJ

SQLJ allows developers to use user-defined datatypes in Java programs. Developers can use JPublisher to map Oracle object and collection types into Java classes to be used in the application.

SQLJ provides access to server objects using SQL statements embedded in the Java code. SQLJ provides compile-time type checking of object types and collections in the SQL statements.

The syntax is based on an ANSI standard (SQLJ Consortium).

SQLJ Object Types

Oracle9*i*, Release 1 (9.0.1), lets you specify Java classes as SQL user-defined object types. You can define columns or rows of this SQLJ type. You can also query and manipulate the objects of this type as if they were SQL primitive types.

Additionally, you can do the following:

- Make the static fields of a class visible in SQL
- Allow the user to call a Java constructor
- Maintain the dependency between the Java class and its corresponding type

See Also:

- *Oracle9i SQL Reference*
- *Oracle9i Application Developer's Guide - Object-Relational Features*
- *Oracle9i SQLJ Developer's Guide and Reference*

Datatype Evolution

A user-defined datatype can be referenced by any of the following schema objects:

- Table or subtable
- Type or subtype
- Program unit (PL/SQL block): procedure, function, package, trigger
- Indextype
- View (including object view)
- Functional index
- Operator

When any of these objects references a type, either directly or indirectly through another type or subtype, it becomes a dependent object on that type. Whenever a type is modified, all dependent program units, views, operators and indextypes are marked *invalid*. The next time each of these invalid objects is referenced, it is

revalidated, using the new type definition. If it is recompiled successfully, then it becomes valid and can be used again.

When a type has either type or table dependents, altering a type definition becomes more complicated because existing persistent data relies on the current type definition.

Oracle9*i*, Release 1 (9.0.1), adds options to the `ALTER TYPE` and `ALTER TABLE` statements so that you can change an object type and propagate the type change to its dependent types and tables. `ALTER TYPE` lets you add or drop methods and attributes from existing types and optionally propagate the changes to dependent types, tables, and even the table data. You can also modify certain attributes of a type.

See Also:

- *Oracle9i SQL Reference* for details about syntax
- *PL/SQL User's Guide and Reference* for details about type specification and body compilation
- *Oracle9i Application Developer's Guide - Object-Relational Features* for details about managing type versions

15

Object Views

This chapter describes object views. It contains the following major sections:

- Introduction to Object Views
- How Object Views Are Defined
- Use of Object Views
- Updates of Object Views
- View Hierarchies

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features*

Introduction to Object Views

Just as a view is a virtual table, an **object view** is a virtual object table.

Oracle provides object views as an extension of the basic relational view mechanism. By using object views, you can create virtual object tables from data—of either built-in or user-defined types—stored in the columns of relational or object tables in the database.

Object views provide the ability to offer specialized or restricted access to the data and objects in a database. For example, you can use an object view to provide a version of an employee object table that does not have attributes containing sensitive data and does not have a deletion method.

Object views allow the use of relational data in object-oriented applications. They let users:

- Try object-oriented programming techniques without converting existing tables
- Convert data gradually and transparently from relational tables to object-relational tables
- Use legacy RDBMS data with existing object-oriented applications

Advantages of Object Views

Using object views can lead to better performance. Relational data that make up a row of an object view traverse the network as a unit, potentially saving many round trips.

You can fetch relational data into the client-side object cache and map it into C or *C++* structures so 3GL applications can manipulate it just like native structures.

Object views provide a gradual migration path for legacy data. They provide for co-existence of relational and object-oriented applications, and they make it easier to introduce object-oriented applications to existing relational data without having to make a drastic change from one paradigm to another.

Object views provide the flexibility of looking at the same relational or object data in more than one way. Thus you can use different in-memory object representations for different applications without changing the way you store the data in the database.

How Object Views Are Defined

Conceptually, the process of defining an object view is simple. It consists of the following actions:

- Defining an object type to be represented by rows of the object view.
- Writing a query that specifies which data in which relational tables contain the attributes for objects of that type.
- Specifying an object identifier, based on attributes of the underlying data, to allow REFs to the objects (rows) of the object view.

The object identifier corresponds to the unique object identifier that Oracle generates automatically for rows of object tables. In the case of object views, however, the declaration must specify something that is unique in the underlying data (for example, a primary key).

If the object view is based on a table or another object view and you do not specify an object identifier, Oracle uses the object identifier from the original table or object view.

If you want to be able to update a complex object view, you may have to take another action:

- Write an INSTEAD OF trigger procedure for Oracle to execute whenever an application program tries to update data in the object view.

After doing these four things, you can use an object view just like an object table.

For example, the following SQL statements define an object view:

```
CREATE TABLE emp_table  (
    empnum   NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

CREATE TYPE employee_t  AS OBJECT(
    empno    NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

CREATE VIEW emp_view1 OF employee_t
    WITH OBJECT OID (empno) AS
        SELECT   e.empnum, e.ename, e.salary, e.job
```

```
FROM     emp_table e
WHERE    job = 'Developer';
```

The object view looks to the user like an object table whose underlying type is `employee_t`. Each row contains an object of type `employee_t`. Each row has a unique object identifier.

Oracle constructs the object identifier based on the specified key. In most cases, it is the primary key of the base table. If the query that defines the object view involves joins, however, you must provide a key across all tables involved in the joins, so that the key still uniquely identifies rows of the object view.

Note: Columns in the `WITH OBJECT OID` clause (`empno` in the example) must also be attributes of the underlying object type (`employee_t` in the example). This makes it easy for trigger programs to identify the corresponding row in the base table uniquely.

See Also:

- *Oracle9i Database Administrator's Guide* for specific directions for defining object views
- "Updates of Object Views" on page 15-5 for more information about writing an `INSTEAD OF` trigger

Use of Object Views

Data in the rows of an object view can come from more than one table, but the object still traverses the network in one operation. When the instance is in the client side object cache, it appears to the programmer as a C or C++ structure or as a PL/SQL object variable. You can manipulate it like any other native structure.

You can refer to object views in SQL statements the same way you refer to an object table. For example, object views can appear in a `SELECT` list, in an `UPDATE SET` clause, or in a `WHERE` clause. You can also define object views on object views.

You can access object view data on the client side using the same OCI calls you use for objects from object tables. For example, you can use `OCIObjectPin()` for pinning a `REF` and `OCIObjectFlush()` for flushing an object to the server. When you update or flush to the server an object in an object view, Oracle updates the object view.

See Also: *Oracle Call Interface Programmer's Guide* for more information about OCI calls

Updates of Object Views

You can update, insert, and delete the data in an object view using the same SQL DML you use for object tables. Oracle updates the base tables of the object view if there is no ambiguity.

A view is not updatable if its view query contains joins, set operators, aggregate functions, `GROUP BY`, or `DISTINCT`. If a view query contains pseudocolumns or expressions, the corresponding view columns are not updatable. Object views often involve joins.

To overcome these obstacles Oracle provides **INSTEAD OF triggers**. They are called `INSTEAD OF` triggers because Oracle executes the trigger body instead of the actual DML statement.

`INSTEAD OF` triggers provide a transparent way to update object views or relational views. You write the same SQL DML (`INSERT`, `DELETE`, and `UPDATE`) statements as for an object table. Oracle invokes the appropriate trigger instead of the SQL statement, and the actions specified in the trigger body take place.

See Also:

- *Oracle9i Application Developer's Guide - Object-Relational Features* for a purchase order/line item example that uses an `INSTEAD OF` trigger
- Chapter 18, "Triggers"

Updates of Nested Table Columns in Views

A nested table can be modified by inserting new elements and updating or deleting existing elements. Nested table columns that are virtual or synthesized, as in a view, are not usually updatable. To overcome this, Oracle allows `INSTEAD OF` triggers to be created on these columns.

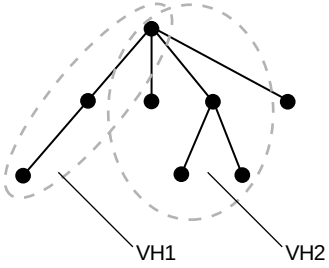
The `INSTEAD OF` trigger defined on a nested table column of a view is fired when the column is modified. If the entire collection is replaced by an update of the parent row, then the `INSTEAD OF` trigger on the nested table column is not fired.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for a purchase order/line item example that uses an `INSTEAD OF` trigger on a nested table column

View Hierarchies

An object view can be created as a subview of another object view. The type of the superview must be the immediate supertype of the type of the object view being created. Thus, you can build an object view hierarchy which has a one-to-one correspondence to the type hierarchy. This does not imply that every view hierarchy must span the entire corresponding type hierarchy. The view hierarchy can be rooted at any subtype of the type hierarchy. Furthermore, it does not have to encompass the entire subhierarchy.

Figure 15–1 Multiple View Hierarchies



By default, the rows of an object view in a view hierarchy include all the rows of all its subviews (direct and indirect) projected over the columns of the given view.

Only one object view can be created as a subview of a given view corresponding to the given subtype; that is, the same view cannot participate in many different view hierarchies. An object view can be created as a subview of only one superview; multiple inheritance is not supported.

The subview inherits the object identifier (OID) from its superview and cannot be explicitly specified in any subview.

Example of Creating a View Hierarchy

```
CREATE VIEW person_v OF person_t
  WITH OBJECT ID(ssn) AS
  SELECT ssn, name, address FROM allpersons WHERE typeid = 1;
```

```
CREATE VIEW student_v of student_t UNDER person_v AS
  SELECT ssn, name, address, deptid, major FROM allpersons
  WHERE typeid = 2;
```

The query underlying a view determines if the view is inherently updatable. This applies to subviews as well. Instead-of triggers can be defined on subviews to perform appropriate DML actions.

Instead-of triggers are **not** inherited by subviews.

Part V

Data Access

Part V describes how to use transactions consisting of SQL statements to access data in an Oracle database. It also describes the procedural language constructs that provide additional functionality for data access.

Part V contains the following chapters:

- Chapter 16, "SQL, PL/SQL, and Java"
- Chapter 17, "Transaction Management"
- Chapter 18, "Triggers"
- Chapter 19, "Dependencies Among Schema Objects"

16

SQL, PL/SQL, and Java

This chapter provides an overview of the Structured Query Language (SQL), PL/SQL, Oracle's procedural extension to SQL, and Java. The chapter includes:

- Introduction to Structured Query Language
- SQL Processing
- The Optimizer
- PL/SQL Overview
- PL/SQL Server Pages
- PL/SQL Program Units
- Procedures and Functions Overview
- Packages Overview
- Java Overview
- Java Program Units

See Also:

- *Oracle9i SQL Reference*
- *PL/SQL User's Guide and Reference*

Introduction to Structured Query Language

SQL is a database access, nonprocedural language. Users describe in SQL what they want done, and the SQL language compiler automatically generates a procedure to navigate the database and perform the desired task.

IBM Research developed and defined SQL, and ANSI/ISO has refined SQL as the standard language for relational database management systems.The minimal conformance level for SQL-99 is known as Core. Core SQL-99 is a superset of SQL-92 Entry Level specification. Oracle9*i* is broadly compatible with the SQL-99 Core specification.

Oracle SQL includes many extensions to the ANSI/ISO standard SQL language, and Oracle tools and applications provide additional statements. The Oracle tools SQL*Plus and Oracle Enterprise Manager allow you to execute any ANSI/ISO standard SQL statement against an Oracle database, as well as additional statements or functions that are available for those tools.

Oracle SQLJ allows applications programmers to embed static SQL operations in Java code in a way that is compatible with the Java design philosophy. A SQLJ program is a Java program containing embedded static SQL statements that comply with the ANSI-standard SQLJ Language Reference syntax.

Although some Oracle tools and applications simplify or mask SQL use, all database operations are performed using SQL. Any other data access method circumvents the security built into Oracle and potentially compromise data security and integrity.

See Also:

- *Oracle9i SQL Reference* for detailed information about SQL statements and other parts of SQL (such as operators, functions, and format models)
- *Oracle Enterprise Manager Administrator's Guide* for information about Oracle Enterprise Manager
- *SQL*Plus User's Guide and Reference* for SQL*Plus statements, including their distinction from SQL statements
- *Oracle9i SQLJ Developer's Guide and Reference* for information about embedding SQL operations in Java code

SQL Statements Overview

All operations performed on the information in an Oracle database are executed using SQL **statements**. A statement consists partially of SQL **reserved words**, which have special meaning in SQL and cannot be used for any other purpose. For example, `SELECT` and `UPDATE` are reserved words and cannot be used as table names.

A SQL statement is a computer program or instruction. The statement must be the equivalent of a complete SQL sentence, as in:

```
SELECT ename, deptno FROM emp;
```

Only a complete SQL statement can be executed, whereas a fragment such as the following generates an error indicating that more text is required before a SQL statement can execute:

```
SELECT ename
```

Oracle SQL statements are divided into the following categories:

- Data manipulation language (DML) statements
- Data definition language statements (DDL)
- Transaction control statements
- Session control statements
- System control statements
- Embedded SQL statements

> **See Also:** Chapter 18, "Triggers" for more information about using SQL statements in PL/SQL program units

Data Manipulation Language Statements Description

Data manipulation language (DML) statements query or manipulate data in existing schema objects. They enable you to:

- Retrieve data from one or more tables or views (`SELECT`); in Oracle9*i*, Release 1 (9.0.1), fetches can be scrollable (see "Scrollable Cursors" on page 16-7)
- Add new rows of data into a table or view (`INSERT`)
- 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)
- See the execution plan for a SQL statement (EXPLAIN PLAN)
- Lock a table or view, temporarily limiting other users' access (LOCK TABLE)

DML statements are the most frequently used SQL statements. Some examples of DML statements are:

```
SELECT ename, mgr, comm + sal FROM emp;

INSERT INTO emp VALUES
    (1234, 'DAVIS', 'SALESMAN', 7698, '14-FEB-1988', 1600, 500, 30);

DELETE FROM emp WHERE ename IN ('WARD','JONES');
```

Data Definition Language Statements Description

Data definition language (DDL) statements define, alter the structure of, and drop schema objects. DDL statements enable you to:

- Create, alter, and drop schema objects and other database structures, including the database itself and database users (CREATE, ALTER, DROP)
- Change the names of schema objects (RENAME)
- Delete all the data in schema objects without removing the objects' structure (TRUNCATE)
- 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 statements implicitly commit the preceding and start a new transaction. Some examples of DDL statements are:

```
CREATE TABLE plants
    (COMMON_NAME VARCHAR2 (15), LATIN_NAME VARCHAR2 (40));

DROP TABLE plants;

GRANT SELECT ON emp TO scott;

REVOKE DELETE ON emp FROM scott;
```

See Also:

- Chapter 24, "Controlling Database Access"
- Chapter 25, "Privileges, Roles, and Security Policies"
- Chapter 26, "Auditing"

for more information about DDL statements that correspond to database access

Transaction Control Statements Description

Transaction control statements manage the changes made by DML statements and group DML statements into transactions. They enable you to:

- Make a transaction's changes permanent (`COMMIT`)
- Undo the changes in a transaction, either since the transaction started or since a savepoint (`ROLLBACK`)
- Set a point to which you can roll back (`SAVEPOINT`)
- Establish properties for a transaction (`SET TRANSACTION`)

Session Control Statements Description

Session control statements manage the properties of a particular user's session. For example, they enable you to:

- Alter the current session by performing a specialized function, such as enabling and disabling the SQL trace facility (`ALTER SESSION`)
- Enable and disable roles (groups of privileges) for the current session (`SET ROLE`)

System Control Statements Description

System control statements change the properties of the Oracle server instance. The only system control statement is `ALTER SYSTEM`. It enables you to change settings (such as the minimum number of shared servers), kill a session, and perform other tasks.

Embedded SQL Statements Description

Embedded SQL statements incorporate DDL, DML, and transaction control statements within a procedural language program. They are used with the Oracle precompilers. Embedded SQL statements enable you to:

- Define, allocate, and release cursors (`DECLARE CURSOR, OPEN, CLOSE`)
- Specify a database and connect to Oracle (`DECLARE DATABASE, CONNECT`)
- Assign variable names (`DECLARE STATEMENT`)
- Initialize descriptors (`DESCRIBE`)
- Specify how error and warning conditions are handled (`WHENEVER`)
- Parse and execute SQL statements (`PREPARE, EXECUTE, EXECUTE IMMEDIATE`)
- Retrieve data from the database (`FETCH`)

Identification of Nonstandard SQL

Oracle provides extensions to the standard SQL Database Language with Integrity Enhancement. The Federal Information Processing Standard for SQL (FIPS 127-2) requires vendors to supply a method for identifying SQL statements that use such extensions. You can identify or **flag** Oracle extensions in interactive SQL, the Oracle precompilers, or SQL*Module by using the FIPS flagger.

If you are concerned with the portability of your applications to other implementations of SQL, use the FIPS flagger.

See Also:

- *Pro*C/C++ Precompiler Programmer's Guide*
- *Pro*COBOL Precompiler Programmer's Guide*
- *SQL*Module for Ada Programmer's Guide*

Recursive SQL

When a DDL statement is issued, Oracle implicitly issues **recursive SQL statements** that modify data dictionary information. Users need not be concerned with the recursive SQL internally performed by Oracle.

Cursors Overview

A **cursor** is a handle or name for a **private SQL area**—an area in memory in which a parsed statement and other information for processing the statement are kept.

Although most Oracle users rely on the automatic cursor handling of the Oracle utilities, the programmatic interfaces offer application designers more control over

cursors. In application development, a cursor is a named resource available to a program and can be used specifically to parse SQL statements embedded within the application.

Each user session can open multiple cursors up to the limit set by the initialization parameter `OPEN_CURSORS`. However, applications should close unneeded cursors to conserve system memory. If a cursor cannot be opened due to a limit on the number of cursors, then the database administrator can alter the `OPEN_CURSORS` initialization parameter.

Some statements (primarily DDL statements) require Oracle to implicitly issue recursive SQL statements, which also require **recursive cursors**. For example, a `CREATE TABLE` statement causes many updates to various data dictionary tables to record the new table and columns. **Recursive calls** are made for those recursive cursors; one cursor can execute several recursive calls. These recursive cursors also use **shared SQL areas**.

Scrollable Cursors

Execution of a cursor puts the results of the query into a set of rows called the result set, which can be fetched sequentially or nonsequentially. **Scrollable cursors** are cursors in which fetches and DML operations do not need to be forward sequential only. Interfaces exist to fetch previously fetched rows, to fetch the nth row in the result set, and to fetch the nth row from the current position in the result set.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about using scrollable cursors in OCI

Shared SQL

Oracle automatically notices when applications send similar SQL statements to the database. The SQL area used to process the first occurrence of the statement is **shared**—that is, used for processing subsequent occurrences of that same statement. Therefore, only one shared SQL area exists for a unique statement. Because shared SQL areas are shared memory areas, any Oracle process can use a shared SQL area. The sharing of SQL areas reduces memory usage on the database server, thereby increasing system throughput.

In evaluating whether statements are similar or identical, Oracle considers SQL statements issued directly by users and applications as well as recursive SQL statements issued internally by a DDL statement.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* and *Oracle9i Database Performance Guide and Reference* for more information about shared SQL

Parsing

Parsing is one stage in the processing of a SQL statement. When an application issues a SQL statement, the application makes a parse call to Oracle. During the parse call, Oracle:

- Checks the statement for syntactic and semantic validity
- Determines whether the process issuing the statement has privileges to execute it
- Allocates a private SQL area for the statement

Oracle also determines whether there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and executes the statement immediately. If not, Oracle generates the parsed representation of the statement, and the user process allocates a shared SQL area for the statement in the library cache and stores its parsed representation there.

Note the difference between an application making a parse call for a SQL statement and Oracle actually parsing the statement. A **parse call** by the **application** associates a SQL statement with a private SQL area. After a statement has been associated with a private SQL area, it can be executed repeatedly without your application making a parse call. A **parse operation** by Oracle allocates a shared SQL area for a SQL statement. Once a shared SQL area has been allocated for a statement, it can be executed repeatedly without being reparsed.

Both parse calls and parsing can be expensive relative to execution, so users should perform them as seldom as possible.

See Also: "PL/SQL Overview" on page 16-18

SQL Processing

This section introduces the basics of SQL processing. Topics include:

- Overview of SQL Statement Execution
- DML Statement Processing
- DDL Statement Processing

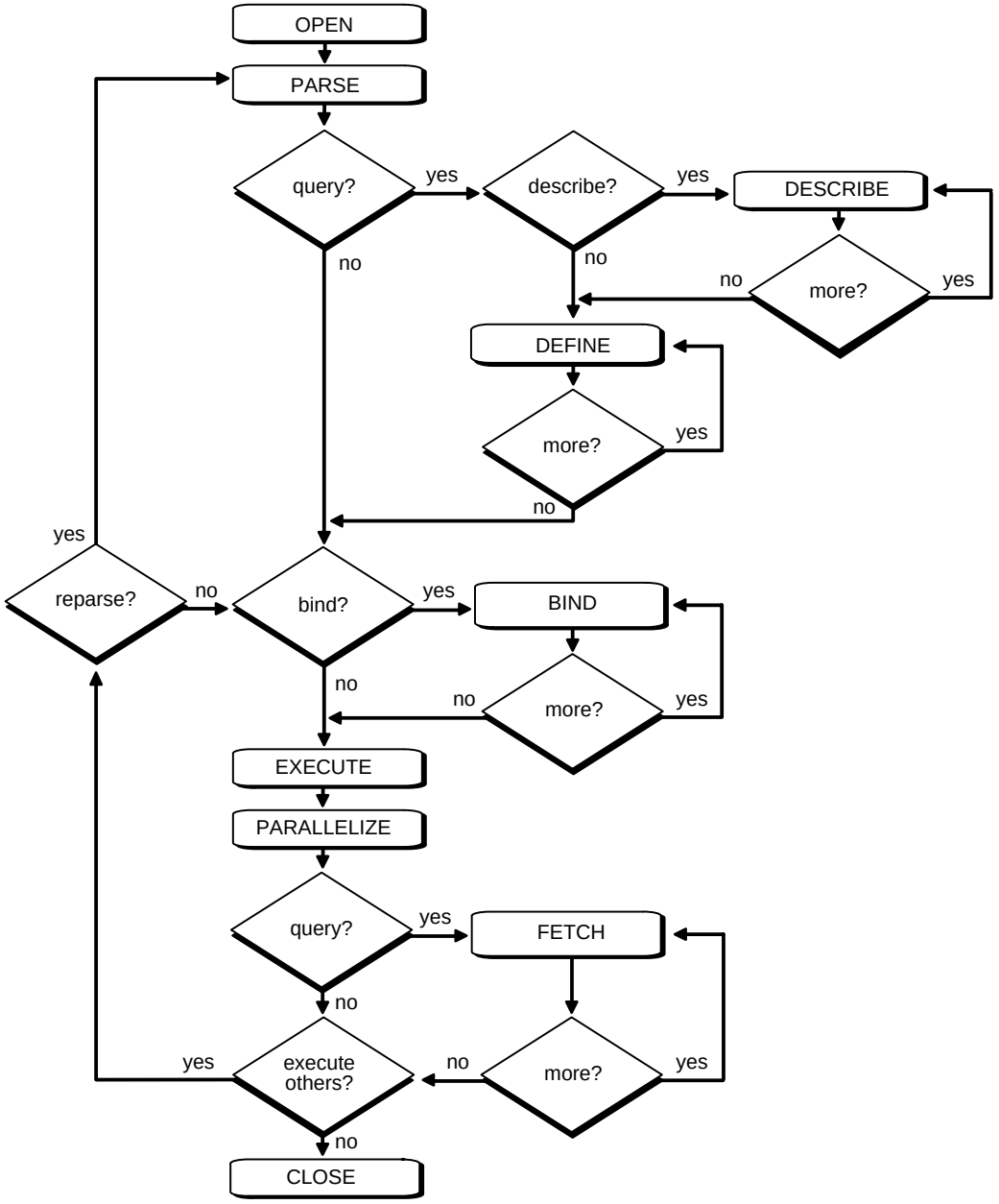
- Control of Transactions

Overview of SQL Statement Execution

Figure 16–1 outlines the stages commonly used to process and execute a SQL statement. In some cases, Oracle can execute these stages in a slightly different order. For example, the `DEFINE` stage could occur just before the `FETCH` stage, depending on how you wrote your code.

For many Oracle tools, several of the stages are performed automatically. Most users need not be concerned with or aware of this level of detail. However, you may find this information useful when writing Oracle applications.

Figure 16–1 The Stages in Processing a SQL Statement



DML Statement Processing

This section provides an example of what happens during the execution of a SQL statement in each stage of DML statement processing.

Assume that you are using a Pro*C program to increase the salary for all employees in a department. Also assume that the program you are using has connected to Oracle and that you are connected to the proper schema to update the `EMP` table. You can embed the following SQL statement in your program:

```
EXEC SQL UPDATE emp SET sal = 1.10 * sal
    WHERE deptno = :dept_number;
```

`DEPT_NUMBER` is a program variable containing a value for department number. When the SQL statement is executed, the value of `DEPT_NUMBER` is used, as provided by the application program.

The following stages are necessary for each type of statement processing:

- Stage 1: Create a Cursor
- Stage 2: Parse the Statement
- Stage 5: Bind Any Variables
- Stage 7: Execute the Statement
- Stage 9: Close the Cursor

Optionally, you can include another stage:

- Stage 6: Parallelize the Statement

Queries (`SELECT`s) require several additional stages, as shown in Figure 16–1:

- Stage 3: Describe Results of a Query
- Stage 4: Define Output of a Query
- Stage 8: Fetch Rows of a Query

See Also: "Query Processing" on page 16-13

Stage 1: Create a Cursor

A program interface call creates a cursor. The cursor is created independent of any SQL statement: it is created in expectation of any SQL statement. In most applications, cursor creation is automatic. However, in precompiler programs, cursor creation can either occur implicitly or be explicitly declared.

Stage 2: Parse the Statement

During parsing, the SQL statement is passed from the user process to Oracle, and a parsed representation of the SQL statement is loaded into a shared SQL area. Many errors can be caught during this stage of statement processing.

Parsing is the process of:

- Translating a SQL statement, verifying it to be a valid statement
- Performing data dictionary lookups to check table and column definitions
- Acquiring parse locks on required objects so that their definitions do not change during the statement's parsing
- Checking privileges to access referenced schema objects
- Determining the optimal execution plan for the statement
- Loading it into a shared SQL area
- Routing all or part of distributed statements to remote nodes that contain referenced data

Oracle parses a SQL statement only if a shared SQL area for an similar SQL statement does not exist in the shared pool. In this case, a new shared SQL area is allocated, and the statement is parsed.

The parse stage includes processing requirements that need to be done only once no matter how many times the statement is executed. Oracle translates each SQL statement only once, reexecuting that parsed statement during subsequent references to the statement.

Although parsing a SQL statement validates that statement, parsing only identifies errors that can be found *before statement execution*. Thus, some errors cannot be caught by parsing. For example, errors in data conversion or errors in data (such as an attempt to enter duplicate values in a primary key) and deadlocks are all errors or situations that can be encountered and reported only during the execution stage.

> **See Also:** "Shared SQL" on page 16-7 for more information about shared SQL areas

Query Processing

Queries are different from other types of SQL statements because, if successful, they return data as results. Whereas other statements simply return success or failure, a query can return one row or thousands of rows. The results of a query are *always in* **tabular format,** and the rows of the result are **fetched** (retrieved), either a row at a time or in groups.

Several issues relate only to query processing. Queries include not only explicit SELECT statements but also the implicit queries *(*subqueries*)* in other SQL statements. For example, each of the following statements requires a query as a part of its execution:

```
INSERT INTO table SELECT...

UPDATE table SET x = y WHERE...

DELETE FROM table WHERE...

CREATE table AS SELECT...
```

In particular, queries:

- Require **read consistency**
- Can use temporary segments for intermediate processing
- Can require the describe, define, and fetch stages of SQL statement processing.

Stage 3: Describe Results of a Query

The describe stage is necessary only if the characteristics of a query's result are not known; for example, when a query is entered interactively by a user.

In this case, the describe stage determines the characteristics (datatypes, lengths, and names) of a query's result.

Stage 4: Define Output of a Query

In the define stage for queries, you specify the location, size, and datatype of variables defined to receive each fetched value. Oracle performs datatype conversion if necessary.

Stage 5: Bind Any Variables

At this point, Oracle knows the meaning of the SQL statement but still does not have enough information to execute the statement. Oracle needs values for any

variables listed in the statement; in the example, Oracle needs a value for DEPT_NUMBER. The process of obtaining these values is called **binding variables**.

A program must specify the location (memory address) where the value can be found. End users of applications may be unaware that they are specifying bind variables, because the Oracle utility can simply prompt them for a new value.

Because you specify the location (binding by reference), you need not rebind the variable before reexecution. You can change its value and Oracle looks up the value on each execution, using the memory address.

You must also specify a datatype and length for each value (unless they are implied or defaulted) if Oracle needs to perform datatype conversion.

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide* (see "Dynamic SQL Method 4")
- *Pro*COBOL Precompiler Programmer's Guide* (see "Dynamic SQL Method 4")

for more information about specifying a datatype and length for a value

Stage 6: Parallelize the Statement

Oracle can parallelize queries (SELECTs, INSERTs, UPDATEs, MERGEs, DELETEs), and some DDL operations such as index creation, creating a table with a subquery, and operations on partitions. Parallelization causes multiple server processes to perform the work of the SQL statement so it can complete faster.

See Also: Chapter 20, "Parallel Execution of SQL Statements" for more information about parallel SQL

Stage 7: Execute the Statement

At this point, Oracle has all necessary information and resources, so the statement is executed. If the statement is a query or an INSERT statement, no rows need to be locked because no data is being changed. If the statement is an UPDATE or DELETE statement, however, all rows that the statement affects are locked from use by other users of the database until the next COMMIT, ROLLBACK, or SAVEPOINT for the transaction. This ensures data integrity.

For some statements you can specify a number of executions to be performed. This is called **array processing**. Given *n* number of executions, the bind and define locations are assumed to be the beginning of an array of size *n*.

Stage 8: Fetch Rows of a Query

In the fetch stage, rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result until the last row has been fetched.

Stage 9: Close the Cursor

The final stage of processing a SQL statement is closing the cursor.

DDL Statement Processing

The execution of DDL statements differs from the execution of DML statements and queries, because the success of a DDL statement requires write access to the data dictionary. For these statements, parsing (Stage 2) actually includes parsing, data dictionary lookup, and execution.

Transaction management, session management, and system management SQL statements are processed using the parse and execute stages. To reexecute them, simply perform another execute.

Control of Transactions

In general, only application designers using the programming interfaces to Oracle are concerned with the types of actions that should be grouped together as one transaction. Transactions must be defined so that work is accomplished in logical units and data is kept consistent. A transaction should consist of all of the necessary parts for one logical unit of work, no more and no less.

- Data in all referenced tables should be in a consistent state before the transaction begins and after it ends.
- Transactions should consist of only the SQL statements that make one consistent change to the data.

For example, a transfer of funds between two accounts (the transaction or logical unit of work) should include the debit to one account (one SQL statement) and the credit to another account (one SQL statement). Both actions should either fail or succeed together as a unit of work; the credit should not be committed without the

debit. Other unrelated actions, such as a new deposit to one account, should not be included in the transfer of funds transaction.

In addition to determining which types of actions form a transaction, when you design an application you must also determine when it is useful to use the `BEGIN_DISCRETE_TRANSACTION` procedure to improve the performance of short, non-distributed transactions.

See Also: "Discrete Transaction Management" on page 17-11

The Optimizer

The optimizer determines the most efficient way to execute a SQL statement. This is an important step in the processing of any data manipulation language (DML) statement: `SELECT, INSERT, UPDATE, MERGE,` or `DELETE.` There are often many different ways to execute a SQL statement; for example, by varying the order in which tables or indexes are accessed. The procedure Oracle uses to execute a statement can greatly affect how quickly the statement executes. The optimizer considers many factors among alternative access paths. It can use either a cost-based or a rule-based approach. In general, always use the cost-based approach. The rule-based approach is available for the benefit of existing applications.

Note: The optimizer might not make the same decisions from one version of Oracle to the next. In recent versions, the optimizer might make different decisions based on better information available to it.

You can influence the optimizer's choices by setting the optimizer approach and goal. You can also gather statistics for the cost-based optimizer (CBO), using Oracle9*i*'s PL/SQL package `DBMS_STATS.`

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 execute a SQL statement. The application designer can use hints in SQL statements to specify how the statement should be executed.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about using `DBMS_STATS`
- *Oracle9i Database Performance Guide and Reference* for more information about the cost-based optimizer, the rule-based optimizer, and the extensible optimizer

Execution Plans

To execute a DML statement, Oracle might need to perform many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of the steps Oracle uses to execute a statement is called an execution plan. An execution plan includes an access method for each table that the statement accesses and an ordering of the tables (the join order). The steps of the execution plan are not performed in the order in which they are numbered.

See Also: *Oracle9i Database Performance Guide and Reference*

Stored Outlines

Stored outlines are abstractions of an execution plan generated by the optimizer at the time the outline was created and are represented primarily as a set of hints. When the outline is subsequently used, these hints are applied at various stages of compilation. Outline data is stored in the `OUTLN` schema.

Oracle9*i*, Release 1 (9.0.1), enables users to tune execution plans by editing stored outlines.

Editing Stored Outlines

The outline is cloned into the user's schema at the onset of the outline editing session. All subsequent editing operations are performed on that clone until the user is satisfied with the edits and chooses to publicize them. In this way, any editing done by the user does not impact the rest of the user community, which would continue to use the public version of the outline until the edits are explicitly saved.

See Also: *Oracle9i Database Performance Guide and Reference* for details about cloning the outline, editing the outline, validating the edits, and publicizing the edits

PL/SQL Overview

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL enables you to mix SQL statements with procedural constructs. With PL/SQL, you can define and execute PL/SQL program units such as procedures, functions, and packages.

PL/SQL program units generally are categorized as anonymous blocks and stored procedures.

An **anonymous block** is a PL/SQL block that appears within your application and it is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear.

A **stored procedure** is a PL/SQL block that Oracle stores in the database and can be called by name from an application. When you create a stored procedure, Oracle parses the procedure and stores its parsed representation in the database. Oracle also allows you to create and store functions (which are similar to procedures) and packages (which are groups of procedures and functions).

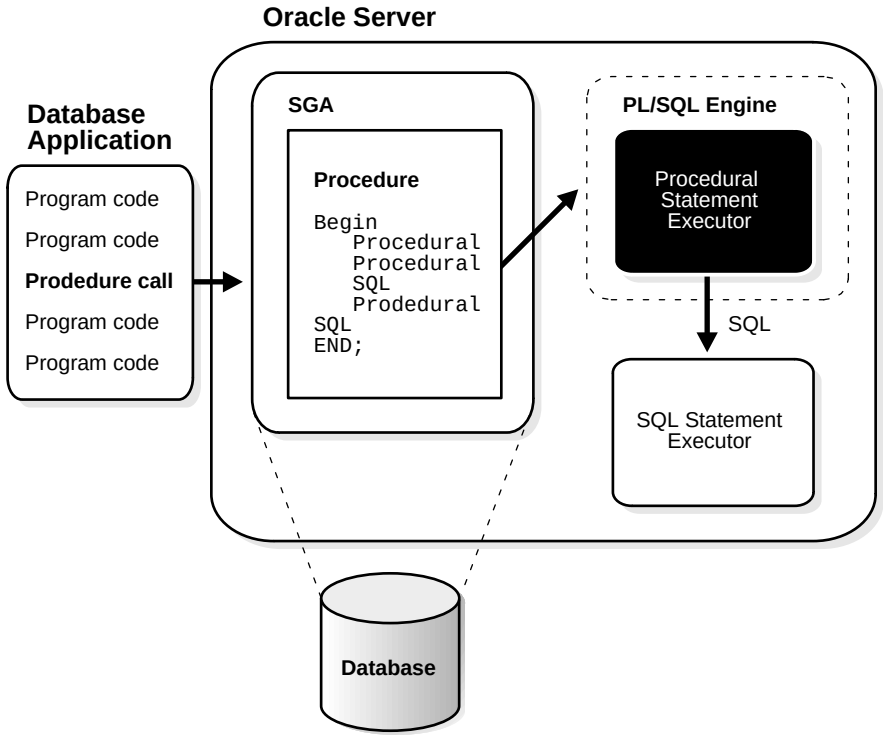
See Also: Chapter 18, "Triggers"

How PL/SQL Executes

The PL/SQL engine, which processes PL/SQL program units, is a component of many Oracle products, including the Oracle server.

Figure 16–2 illustrates the PL/SQL engine contained in Oracle server.

Figure 16–2 The PL/SQL Engine and the Oracle Server



The program unit is stored in a database. When an application calls a procedure stored in the database, Oracle loads the compiled program unit into the shared pool in the system global area (SGA). The PL/SQL and SQL statement executors work together to process the statements within the procedure.

The following Oracle products contain a PL/SQL engine:

- Oracle server
- Oracle Forms (Version 3 and later)
- SQL*Menu (Version 5 and later)
- Oracle Reports (Version 2 and later)
- Oracle Graphics (Version 2 and later)

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 (Version 3 or later).

Also, you can pass anonymous blocks to Oracle from applications developed with these tools:

- Oracle precompilers (including user exits)
- Oracle Call Interfaces (OCIs)
- SQL*Plus
- Oracle Enterprise Manager

Language Constructs for PL/SQL

PL/SQL blocks can include the following PL/SQL language constructs:

- Variables and constants
- Cursors
- Exceptions

This section gives a general description of each construct.

> **See Also:** *PL/SQL User's Guide and Reference*

Variables and Constants

Variables and constants can be declared within a procedure, function, or package. A variable or constant can be used in a SQL or PL/SQL statement to capture or provide a value when one is needed.

> **Note:** Some interactive tools, such as SQL*Plus, allow you to define variables in your current session. You can use such variables just as you would variables declared within procedures or packages.

Cursors

Cursors can be declared explicitly within a procedure, function, or package to facilitate record-oriented processing of Oracle data. Cursors also can be declared implicitly (to support other data manipulation actions) by the PL/SQL engine.

See Also: "Scrollable Cursors" on page 16-7

Exceptions

PL/SQL allows you to explicitly handle internal and user-defined error conditions, called **exceptions**, that arise during processing of PL/SQL code. Internal exceptions are caused by illegal operations, such as division by zero, or Oracle errors returned to the PL/SQL code. User-defined exceptions are explicitly defined and signaled within the PL/SQL block to control processing of errors specific to the application (for example, debiting an account and leaving a negative balance).

When an exception is raised (signaled), the normal execution of the PL/SQL code stops, and a routine called an exception handler is invoked. Specific exception handlers can be written to handle any internal or user-defined exception.

Stored Procedures

Oracle also allows you to create and call stored procedures. If your application calls a stored procedure, the parsed representation of the procedure is retrieved from the database and processed by the PL/SQL engine in Oracle.

Note: While many Oracle products have PL/SQL components, this manual covers only the procedures and packages that can be stored in an Oracle database and processed using the PL/SQL engine of the Oracle server.

You can call stored procedures from applications developed using these tools:

- Oracle precompilers (including user exits)
- Oracle Call Interfaces (OCIs)
- SQL*Module
- SQL*Plus
- Oracle Enterprise Manager

You can also call a stored procedure from another PL/SQL block, either an anonymous block or another stored procedure.

See Also:

- *Pro*C/C++ Precompiler Programmer's Guide* for information about how to call stored C or C++ procedures
- *Pro*COBOL Precompiler Programmer's Guide* for information about how to call stored COBOL procedures
- Other appropriate programmer's guides for more information about how to call stored procedures of specific kinds of application

Dynamic SQL in PL/SQL

PL/SQL can execute **dynamic SQL** statements whose complete text is not known until runtime. Dynamic SQL statements are stored in character strings that are entered into, or built by, the program at runtime. This enables you to create general purpose procedures. For example, using dynamic SQL allows you to create a procedure that operates on a table whose name is not known until runtime.

You can write stored procedures and anonymous PL/SQL blocks that include dynamic SQL in two ways:

- By embedding dynamic SQL statements in the PL/SQL block
- By using the `DBMS_SQL` package

Additionally, you can issue DML or DDL statements using dynamic SQL. This helps solve the problem of not being able to statically embed DDL statements in PL/SQL. For example, you can choose to issue a `DROP TABLE` statement from within a stored procedure by using the `EXECUTE IMMEDIATE` statement or the `PARSE` procedure supplied with the `DBMS_SQL` package.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for a comparison of the two approaches to dynamic SQL
- *PL/SQL User's Guide and Reference* for details about dynamic SQL
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

PL/SQL Server Pages

PL/SQL Server Pages (PSP) are server-side web pages (in HTML or XML) with embedded PL/SQL scripts marked with special tags. To produce dynamic web

pages, developers have usually written CGI programs in C or Perl that fetch data and produce the entire web page within the same program. The development and maintenance of such dynamic pages is costly and time-consuming.

Scripting fulfills the demand for rapid development of dynamic web pages. Small scripts can be embedded in HTML pages without changing their basic HTML identity. The scripts contain the logic to produce the dynamic portions of HTML pages and are executed when the pages are requested by the users.

The separation of HTML content from application logic makes script pages easier to develop, debug, and maintain. The simpler development model, along the fact that scripting languages usually demand less programming skill, enables web page writers to develop dynamic web pages.

There are two kinds of embedded scripts in HTML pages: client-side scripts and server-side scripts. **Client-side scripts** are returned as part of the HTML page and are executed in the browser. They are mainly used for client-side navigation of HTML pages or data validation. **Server-side scripts**, while also embedded in the HTML pages, are executed on the server side. They fetch and manipulate data and produce HTML content that is returned as part of the page. PSP scripts are server-side scripts.

A PL/SQL gateway receives HTTP requests from an HTTP client, invokes a PL/SQL stored procedure as specified in the URL, and returns the HTTP output to the client. A PL/SQL Server Page is processed by a PSP compiler, which compiles the page into a PL/SQL stored procedure. When the procedure is executed by the gateway, it generates the web page with dynamic content. PSP is built on one of two existing PL/SQL gateways:

- PL/SQL Cartridge of Oracle Application Server
- WebDB

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about PL/SQL Server Pages

PL/SQL Program Units

This section discusses the procedural capabilities of Oracle. It includes the following sections:

- Introduction to Stored Procedures and Packages
- Stored Procedures and Functions
- Packages

> **See Also:** Chapter 19, "Dependencies Among Schema Objects" for information about the dependencies of procedures, functions, and packages, and how Oracle manages these dependencies

Introduction to Stored Procedures and Packages

Oracle allows you to access and manipulate database information using procedural schema objects called **PL/SQL program units**. Procedures, functions, and packages are all examples of PL/SQL program units.

PL/SQL is Oracle's procedural language extension to SQL. It extends SQL with flow control and other statements that make it possible to write complex programs. The **PL/SQL engine** is the tool you use to define, compile, and execute PL/SQL program units. This engine is a special component of many Oracle products, including the Oracle server.

While many Oracle products have PL/SQL components, this chapter specifically covers the procedures and packages that can be stored in an Oracle database and processed using the Oracle server PL/SQL engine. The PL/SQL capabilities of each Oracle tool are described in the appropriate tool's documentation.

> **See Also:** "PL/SQL Overview" on page 16-18

Stored Procedures and Functions

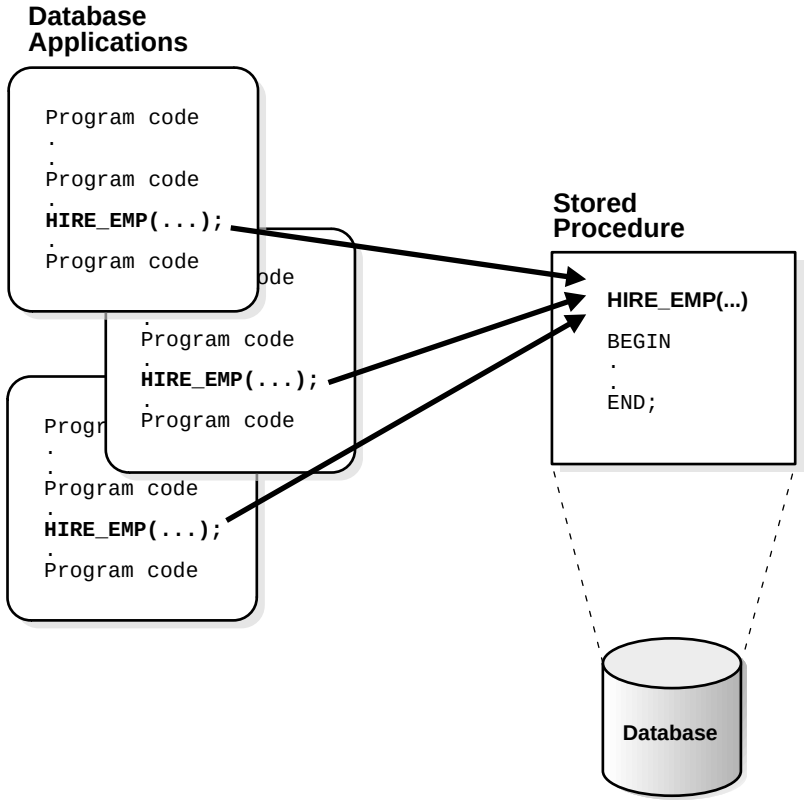
Procedures and functions are schema objects that logically group a set of SQL and other PL/SQL programming language statements together to perform a specific task. Procedures and functions are created in a user's schema and stored in a database for continued use. You can execute a procedure or function interactively by:

- Using an Oracle tool, such as SQL*Plus
- Calling it explicitly in the code of a database application, such as an Oracle Forms or Precompiler application
- Calling it explicitly in the code of another procedure or trigger

Figure 16–3 illustrates a simple procedure that is stored in the database and called by several different database applications.

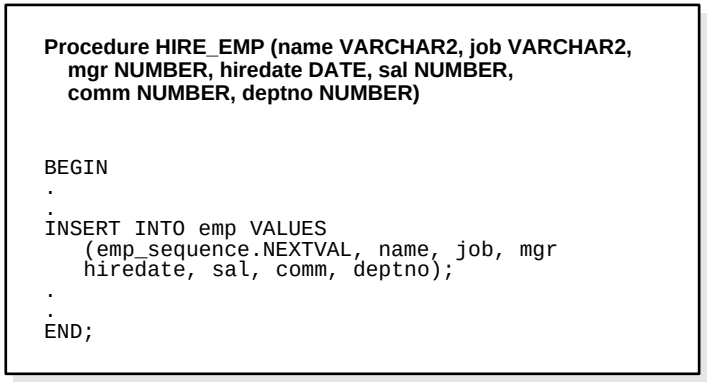
Procedures and functions are identical except that functions always return a single value to the caller, while procedures do not. For simplicity, **procedure** as used in the remainder of this chapter means **procedure** *or* **function**.

Figure 16–3 A Stored Procedure



The stored procedure in Figure 16–3, which inserts an employee record into the EMP table, is shown in Figure 16–4.

Figure 16–4 The HIRE_EMP Procedure



All of the database applications in Figure 16–3 call the `HIRE_EMP` procedure. Alternatively, a privileged user can use Oracle Enterprise Manager or SQL*Plus to execute the `HIRE_EMP` procedure using the following statement:

```
EXECUTE hire_emp ('TSMITH', 'CLERK', 1037, SYSDATE, \
                  500, NULL, 20);
```

This statement places a new employee record for `TSMITH` in the `EMP` table.

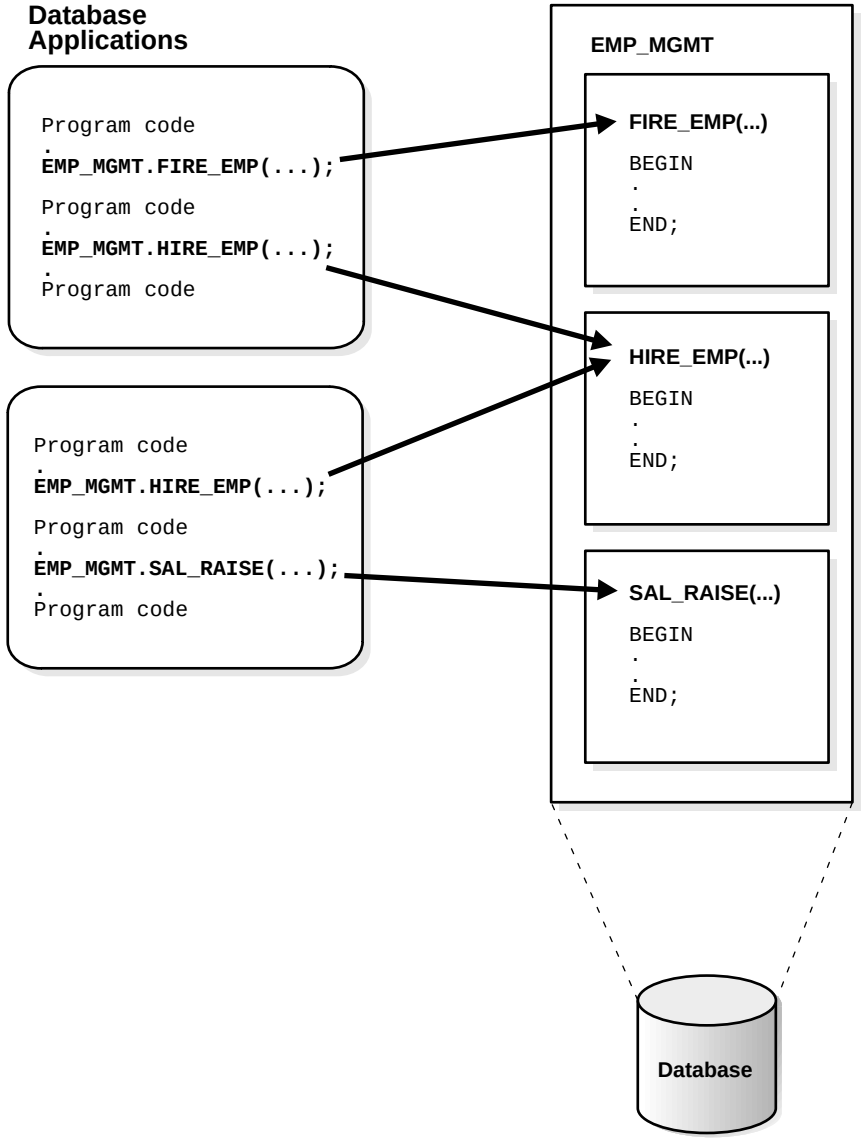
See Also: *PL/SQL User's Guide and Reference*

Packages

A **package** is a group of related procedures and functions, together with the cursors and variables they use, stored together in the database for continued use as a unit. Similar to standalone procedures and functions, packaged procedures and functions can be called explicitly by applications or users.

Figure 16–5 illustrates a package that encapsulates a number of procedures used to manage an employee database.

Figure 16–5 A Stored Package



Database applications explicitly call packaged procedures as necessary. After being

granted the privileges for the EMP_MGMT package, a user can explicitly execute any of the procedures contained in it. For example, Oracle Enterprise Manager or SQL*Plus can issue the following statement to execute the HIRE_EMP package procedure:

```
EXECUTE emp_mgmt.hire_emp ('TSMITH', 'CLERK', 1037, SYSDATE, 500, NULL, 20);
```

Packages offer several development and performance advantages over standalone stored procedures.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

Procedures and Functions Overview

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 executed as a unit to solve a specific problem or perform a set of related tasks. Procedures and functions permit the caller to provide parameters that can be input only, output only, or input and output values. Procedures and functions allow you to combine the ease and flexibility of SQL with the procedural functionality of a structured programming language.

Benefits of Procedures

Procedures provide advantages in the following areas:

- Security with Definer-Rights Procedures
- Inherited Privileges and Schema Context with Invoker-Rights Procedures
- Performance
- Memory Allocation
- Productivity
- Integrity

Security with Definer-Rights Procedures

Stored procedures can help enforce data security. You can restrict the database operations that users can perform by allowing them to access data only through

procedures and functions that execute with the definer's privileges. For example, you can grant users access to a procedure that updates a table but not grant them access to the table itself. When a user invokes the procedure, the procedure executes with the privileges of the procedure's owner. Users who have only the privilege to execute the procedure (but not the privileges to query, update, or delete from the underlying tables) can invoke the procedure, but they cannot manipulate table data in any other way.

> **See Also:** "Dependency Tracking for Stored Procedures" on page 16-31

Inherited Privileges and Schema Context with Invoker-Rights Procedures

An invoker-rights procedure inherits privileges and schema context from the procedure that calls it. In other words, an invoker-rights procedure is not tied to a particular user or schema, and each invocation of an invoker-rights procedure operates in the current user's schema with the current user's privileges. If you are an application developer, invoker-rights procedures make it easy for you to centralize application logic, even when the underlying data is divided among user schemas.

For example, a a user who executes an update procedure on the `EMP` table as a manager can update salary, whereas a user who executes the same procedure as a clerk can be restricted to updating address data.

Performance

Stored procedures can improve database performance in several ways:

- The amount of information that must be sent over a network is small compared with issuing individual SQL statements or sending the text of an entire PL/SQL block to Oracle, because the information is sent only once and thereafter invoked when it is used.
- A procedure's compiled form is readily available in the database, so no compilation is required at execution time.
- If the procedure is already present in the shared pool of the system global area (SGA), retrieval from disk is not required, and execution can begin immediately.

Memory Allocation

Because stored procedures take advantage of the shared memory capabilities of Oracle, only a single copy of the procedure needs to be loaded into memory for

execution by multiple users. Sharing the same code among many users results in a substantial reduction in Oracle memory requirements for applications.

Productivity

Stored procedures increase development productivity. By designing applications around a common set of procedures, you can avoid redundant coding and increase your productivity.

For example, procedures can be written to insert, update, or delete employee records from the `EMP` table. These procedures can then be called by any application without rewriting the SQL statements necessary to accomplish these tasks. If the methods of data management change, only the procedures need to be modified, not all of the applications that use the procedures.

Integrity

Stored procedures improve the integrity and consistency of your applications. By developing all of your applications around a common group of procedures, you can reduce the likelihood of committing coding errors.

For example, you can test a procedure or function to guarantee that it returns an accurate result and, once it is verified, reuse it in any number of applications without testing it again. If the data structures referenced by the procedure are altered in any way, only the procedure needs to be recompiled. Applications that call the procedure do not necessarily require any modifications.

Procedure Guidelines

Use the following guidelines when designing stored procedures:

- Define procedures to complete a single, focused task. Do not define long procedures with several distinct subtasks, because subtasks common to many procedures can be duplicated unnecessarily in the code of several procedures.
- Do not define procedures that duplicate the functionality already provided by other features of Oracle. For example, do not define procedures to enforce simple data integrity rules that you could easily enforce using declarative integrity constraints.

Anonymous PL/SQL Blocks Compared with Stored Procedures

A stored procedure is created and stored in the database as a schema object. Once created and compiled, it is a named object that can be executed without

recompiling. Additionally, dependency information is stored in the data dictionary to guarantee the validity of each stored procedure.

As an alternative to a stored procedure, you can create an anonymous PL/SQL block by sending an unnamed PL/SQL block to the Oracle server from an Oracle tool or an application. Oracle compiles the PL/SQL block and places the compiled version 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. 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.

In either case, moving PL/SQL blocks out of a database application and into database procedures stored either in the database or in memory, you avoid unnecessary procedure recompilations by Oracle at runtime, improving the overall performance of the application and Oracle.

Standalone Procedures

Stored procedures not defined within the context of a package are called **standalone procedures**. Procedures defined within a package are considered a part of the package.

> **See Also:** "Packages Overview" on page 16-33 for information about the advantages of packages

Dependency Tracking for Stored Procedures

A stored procedure depends on the objects referenced in its body. Oracle automatically tracks and manages such dependencies. For example, if you alter the definition of a table referenced by a procedure, the procedure must be recompiled to validate that it will continue to work as designed. Usually, Oracle automatically administers such dependency management.

> **See Also:** Chapter 19, "Dependencies Among Schema Objects" for more information about dependency tracking

External Procedures

A PL/SQL procedure executing on an Oracle server can call an external procedure or function that is written in the C programming language and stored in a shared library. The C routine executes in a separate address space from that of the Oracle server.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for more information about external procedures

Table Functions

Table functions are defined as functions that can produce a set of rows as output. In other words, table functions return a collection type instance (nested table and VARRAY datatypes). Oracle9*i* allows users to define table functions.

Pipelined Table Functions

Oracle9*i*, Release 1 (9.0.1), allows table functions to **pipeline** results (return results iteratively) out of the functions. This can be achieved by either providing an implementation of the `ODCITable` interface, or using native PL/SQL instructions.

Pipelining helps to improving the performance of a number of applications, such as Oracle Warehouse Builder (OWB) and cartridges groups.

The ETL (Extraction-Transformation-Load) process in data warehouse building extracts data from an OLTP system. The extracted data passes through a sequence of transformations (written in procedural languages such as PL/SQL) before it is loaded into a data warehouse.

Parallel Execution of Table Functions

Oracle9*i*, Release 1 (9.0.1), allows parallel execution of table and non-table functions. Parallel execution provides the following extensions:

- Functions can directly accept a set of rows corresponding to a subquery operand.
- A set of input rows can be partitioned among multiple instances of a parallel function.

The function developer specifies how the input rows should be partitioned between parallel instances of the function.

See Also:

- *Oracle9i Data Cartridge Developer's Guide*
- *PL/SQL User's Guide and Reference*

for detailed accounts of table functions

Packages Overview

Packages encapsulate related procedures, functions, and associated cursors and variables together as a unit in the database.

You create a package in two parts: the specification and the body. A package's **specification** declares all public constructs of the package and the **body** defines all constructs (public and private) of the package. This separation of the two parts provides the following advantages:

- The developer has more flexibility in the development cycle. You can create specifications and reference public procedures without actually creating the package body.
- You can alter procedure bodies contained within the package body separately from their publicly declared specifications in the package specification. As long as the procedure specification does not change, objects that reference the altered procedures of the package are never marked invalid. That is, they are never marked as needing recompilation.

Benefits of Packages

Packages are used to define related procedures, variables, and cursors and are often implemented to provide advantages in the following areas:

- Encapsulation of related procedures and variables
- Declaration of public and private procedures, variables, constants, and cursors
- Better performance

Encapsulation Stored packages allow you to **encapsulate** or group stored procedures, variables, datatypes, and so forth in a single named, stored unit in the database. This strategy provides better organization during the development process.

Encapsulation of procedural constructs in a package also makes privilege management easier. Granting the privilege to use a package makes all constructs of the package accessible to the grantee.

Public and Private Data and Procedures The methods of package definition allow you to specify which variables, cursors, and procedures are:

public	Directly accessible to the user of a package
private	Hidden from the user of a package

For example, a package can contain 10 procedures. You can define the package so that only 3 procedures are public and therefore available for execution by a user of the package. The remainder of the 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`.

> **See Also:** Chapter 24, "Controlling Database Access" for more information about grants to `PUBLIC`

Performance Improvement An entire package is loaded into memory when a procedure within 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. Therefore, when calls to related packaged procedures occur, no disk I/O is necessary to execute the compiled code already 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.

Java Overview

Java, which was developed at Sun Microsystems, has emerged over the last several years as the object-oriented programming language of choice. It includes the following concepts:

- A Java virtual machine (JVM), which provides the fundamental basis for platform independence
- Automated storage management techniques, the most visible of which is garbage collection

- Language syntax that borrows from C and enforces strong typing

The result is a language easily learned by existing C programmers, but it remains object-oriented and efficient for application-level programs.

Java and Object-Oriented Programming Terminology

This section covers some basic terminology for discussing details of Java application development in the Oracle9*i* environment.

> **See Also:** *Oracle9i Java Stored Procedures Developer's Guide* for more information about object-oriented programming and for pointers to additional reference material

Classes

All object-oriented programming languages support the concept of a class. As with a table definition, a class provides a template for objects that share common characteristics. Each class can contain the following:

- Attributes—static or instance variables that each object of a particular class possesses.
- Methods—you can invoke methods defined by the class or inherited by any classes extended from the class.

When you create an object from a class, you are creating an instance of that class. The instance contains the fields of an object, which are known as its data, or state.

When you create an instance, the attributes store individual and private information relevant only to the employee. That is, the information contained within an employee instance is known only for that single employee.

Attributes

Attributes within an instance are known as fields. Instance fields are analogous to the fields of a relational table row. The class defines the fields, as well as the type of each field. You can declare fields in Java to be static, public, private, protected, or default access.

- Public, private, protected, or default access fields are created within each instance.
- Static fields are like global variables in that the information is available to all instances of the employee class.

The language specification defines the rules of visibility of data for all fields. Rules of visibility define under what circumstances you can access the data in these fields.

Methods

The class also defines the methods you can invoke on an instance of that class. Methods are written in Java and define the behavior of an object. This bundling of state and behavior is the essence of encapsulation, which is a feature of all object-oriented programming languages. If you define an `Employee` class, declaring that each employee's `id` is a private field, other objects can access that private field only if a method returns the field. In this example, an object could retrieve the employee's identifier by invoking the `Employee.getId()` method.

In addition, with encapsulation, you can declare that the `Employee.getId()` method is private, or you can decide not to write an `Employee.getId()` method. Encapsulation helps you write programs that are reusable and not misused. Encapsulation makes public only those features of an object that are declared public; all other fields and methods are private. Private fields and methods can be used for internal object processing.

Class Hierarchy

Java defines classes within a large hierarchy of classes. At the top of the hierarchy is the `Object` class. All classes in Java inherit from the `Object` class at some level, as you walk up through the inheritance chain of superclasses. When we say Class B inherits from Class A, each instance of Class B contains all the fields defined in class B, as well as all the fields defined in Class A. You can invoke any method on an instance of Class B that was defined in either Class A or B.

Instances of Class B are substitutable for instances of Class A, which makes inheritance another powerful construct of object-oriented languages for improving code reuse. You can create new classes that define behavior and state where it makes sense in the hierarchy, yet make use of preexisting functionality in class libraries.

Interfaces

Java supports only single inheritance; that is, each class has one and only one class from which it inherits. If you must inherit from more than one source, Java provides the equivalent of multiple inheritance, without the complications and confusion that usually accompany it, through interfaces. Interfaces are similar to classes; however, interfaces define method signatures, not implementations. The methods

are implemented in classes declared to implement an interface. Multiple inheritance occurs when a single class simultaneously supports many interfaces.

The Java Virtual Machine (JVM)

As with other high-level computer languages, your Java source compiles to low-level machine instructions. In Java, these instructions are known as bytecodes (because their size is uniformly one byte of storage). Most other languages, such as C, compile to machine-specific instructions, such as instructions specific to an Intel or HP processor. Your Java source compiles to a standard, platform-independent set of bytecodes, which interacts with a Java virtual machine (JVM). In Oracle9*i*, this is known as Aurora. The JVM is a separate program optimized for the specific platform on which you execute your Java code. Your Java source is compiled into bytecodes, which are platform independent. Each platform has installed a JVM that is specific to its operating system. The Java bytecodes from your source get interpreted through the JVM into appropriate platform dependent actions.

When you develop a Java program, you use predefined core class libraries written in the Java language. The Java core class libraries are logically divided into packages that provide commonly-used functionality, such as basic language support (`java.lang`), I/O (`java.io`), and network access (`java.net`). Together, the JVM and core class libraries provide a platform on which Java programmers can develop with the confidence that any hardware and operating system that supports Java will execute their program. This concept is what drives the "write once, run anywhere" idea of Java.

Oracle's Java applications sit on top of the Java core class libraries, which in turn sit on top of the JVM. Because Oracle's Java support system is located within the database, the JVM interacts with the Oracle database libraries, instead of directly with the operating system.

Sun Microsystems furnishes publicly available specifications for both the Java language and the JVM. The Java language specification (JLS) defines things such as syntax and semantics, and the JVM specification defines the necessary low-level behavior for the "machine" that executes the bytecodes. In addition, Sun Microsystems provides a compatibility test suite for JVM implementors to determine if they have complied with the specifications. This test suite is known as the Java Compatibility Kit (JCK). Oracle's JVM implementation complies fully with JCK. Part of the overall Java strategy is that an openly specified standard, together with a simple way to verify compliance with that standard, allows vendors to offer uniform support for Java across all platforms.

Java Program Units

Package DBMS_JAVA

When initializing the JServer, the `initjvm.sql` script creates the PL/SQL package `DBMS_JAVA`. Some entrypoints of `DBMS_JAVA` are for your use, and others are only for internal use. The corresponding Java class `DbmsJava` provides methods for accessing RDBMS functionality from Java.

The `DBMS_JAVA` package supplies the following entrypoints:

```
FUNCTION longname (shortname VARCHAR2) RETURN VARCHAR2
```

Return the full name from a Java schema object. Because Java classes and methods can have names exceeding the maximum SQL identifier length, Aurora uses abbreviated names internally for SQL access. This function simply returns the original Java name for any (potentially) truncated name. An example of this function is to print the fully qualified name of classes that are invalid:

```
select dbms_java.longname (object_name) from user_objects
   where object_type = 'JAVA CLASS' and status = 'INVALID';
```

```
FUNCTION shortname (longname VARCHAR2) RETURN VARCHAR2
```

You can specify a full name to the database by using the `shortname()` routine of the `DBMS_JAVA` package, which takes a full name as input and returns the corresponding short name. This is useful when verifying that your classes loaded by querying the `USER_OBJECTS` view.

> **See Also:** *Oracle9i Java Stored Procedures Developer's Guide* for examples of these functions

```
FUNCTION get_compiler_option(what VARCHAR2, optionName VARCHAR2)
PROCEDURE set_compiler_option(what VARCHAR2, optionName VARCHAR2,
                              value VARCHAR2)
PROCEDURE reset_compiler_option(what VARCHAR2, optionName VARCHAR2)
```

These three entry points control the options of the JServer Java and SQLJ compiler Oracle9*i* delivers.

See Also:

- *Oracle9i Java Developer's Guide* for an example of these options
- *Oracle9i Java Stored Procedures Developer's Guide*
- *Oracle9i SQLJ Developer's Guide and Reference*

```
PROCEDURE set_output (buffersize NUMBER)
```

This procedure redirects the output of Java stored procedures and triggers to the `DBMS_OUTPUT` package.

```
PROCEDURE loadjava(options varchar2)
PROCEDURE loadjava(options varchar2, resolver varchar2)
PROCEDURE dropjava(options varchar2)
```

These procedures allow you to load and drop classes within the database using a call rather than through the `loadjava` or `dropjava` command-line tools. To execute within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
call dbms_java.dropjava('... options...');
```

The options are identical to those specified for the `loadjava` and `dropjava` command-line tools. Separate each option with a blank. Do not separate the options with a comma. The only exception for this is the `loadjava -resolver` option, which contains blanks. For `-resolver,` specify all other options first, separate these options by a comma, and then specify the `-resolver` option with its definition. Do not specify the following options, because they relate to the database connection for the `loadjava` command-line tool: `-thin, -oci8, -user, -password.` The output is directed to `stderr.`

```
PROCEDURE grant_permission( grantee varchar2,
                 permission_type varchar2,
                 permission_name varchar2,
                 permission_action varchar2 )

PROCEDURE restrict_permission( grantee varchar2,
                 permission_type varchar2,
                 permission_name varchar2,
                 permission_action varchar2)

PROCEDURE grant_policy_permission( grantee varchar2,
                 permission_schema varchar2,
                 permission_type varchar2,
```

```
                permission_name varchar2)

PROCEDURE revoke_permission(permission_schema varchar2,
                permission_type varchar2,
                permission_name varchar2,
                permission_action varchar2)

PROCEDURE disable_permission(key number)

PROCEDURE enable_permission(key number)

PROCEDURE delete_permission(key number)
```

These entry points control the JVM permissions.

```
PROCEDURE start_debugging(host varchar2, port number,
                          timeout number)

PROCEDURE stop_debugging

PROCEDURE restart_debugging(timeout number)
```

These entry points start and stop the debug agent when debugging.

17

Transaction Management

This chapter defines a transaction and describes how you can manage your work using transactions. It includes:

- Introduction to Transactions
- Oracle and Transaction Management
- Discrete Transaction Management
- Autonomous Transactions

Introduction to Transactions

A **transaction** is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all **committed** (applied to the database) or all **rolled back** (undone from the database).

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly with a `COMMIT` or `ROLLBACK` statement or implicitly when a DDL statement is issued.

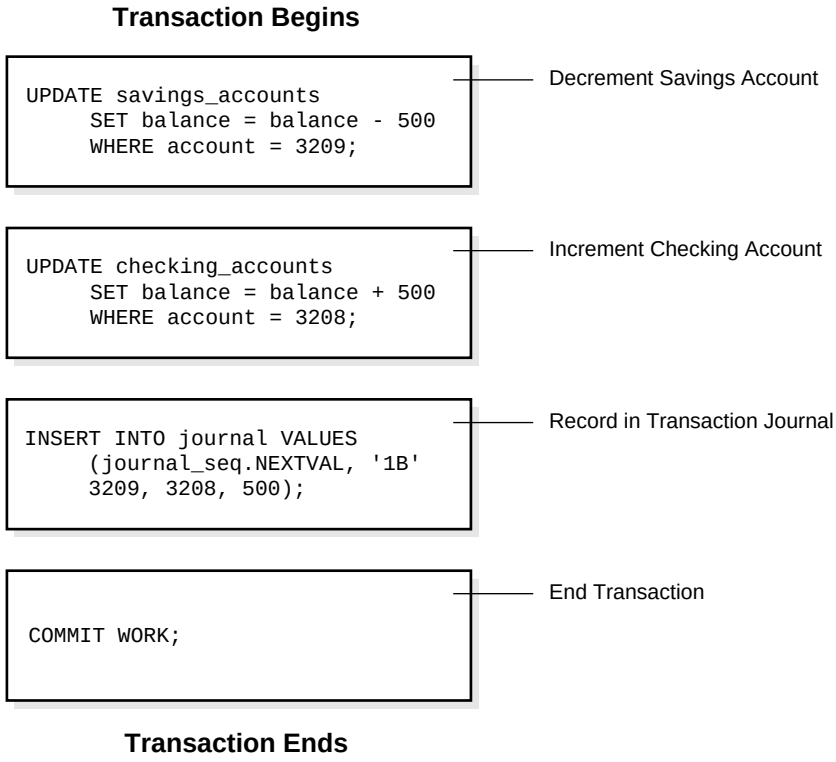
To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:

- Decrement the savings account
- Increment the checking account
- Record the transaction in the transaction journal

Oracle must allow for two situations. If all three SQL statements can be performed to maintain the accounts in proper balance, 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, the entire transaction must be rolled back so that the balance of all accounts is correct.

Figure 17–1 illustrates the banking transaction example.

Figure 17–1 A Banking Transaction



Statement Execution and Transaction Control

A SQL statement that executes successfully is different from a committed transaction. Executing successfully means that a single statement was:

- Parsed
- Found to be a valid SQL construction
- Executed without error as an atomic unit. For example, all rows of a multirow update are changed.

However, until the transaction that contains the statement is committed, the transaction can be rolled back, and all of the changes of the statement can be undone. A statement, rather than a transaction, executes successfully.

Committing means that a user has explicitly or implicitly requested that the changes in the transaction be made permanent. An explicit request means that the user issued a `COMMIT` statement. An implicit request can be made through normal termination of an application or in data definition language, for example. The changes made by the SQL statement(s) of your transaction become permanent and visible to other users only after your transaction has been committed. Only other users' transactions that started after yours will see the committed changes.

Beginning with Oracle9*i*, Release 1 (9.0.1), you can name a transaction by using the `SET TRANSACTION ... NAME` command before you start the transaction. This makes it easier to monitor long-running transactions and to resolve in-doubt distributed transactions.

See Also: "Transaction Naming" on page 17-9

Statement-Level Rollback

If at any time during execution a SQL statement causes an error, all effects of the statement are rolled back. The effect of the rollback is as if that statement had never been executed. This operation is a **statement-level rollback**.

Errors discovered during SQL statement **execution** cause statement-level rollbacks. An example of such an error is attempting to insert a duplicate value in a primary key. Single SQL statements involved in a **deadlock** (competition for the same data) can also cause a statement-level rollback. Errors discovered during SQL statement **parsing**, such as a syntax error, have not yet been executed, so they do not cause a statement-level rollback.

A SQL statement that fails causes the loss only of any work it would have performed itself. *It does not cause the loss of any work that preceded it in the current transaction*. If the statement is a DDL statement, then the implicit commit that immediately preceded it is not undone.

The user can also request a statement-level rollback by issuing a `ROLLBACK` statement.

Note: Users cannot directly refer to implicit savepoints in rollback statements.

See Also: "Deadlocks" on page 22-18

Resumable Space Allocation

Oracle9*i* provides a means for suspending, and later resuming, the execution of large database operations in the event of space allocation failures. This enables an administrator to take corrective action, instead of the Oracle database server returning an error to the user. After the error condition is corrected, the suspended operation automatically resumes. This feature is called **resumable space allocation** and the statements that are affected are called **resumable statements**.

A statement executes in a resumable mode only when the client explicitly enables resumable semantics for the session using the `ALTER SESSION` statement.

A resumable statement is suspended when one of the following conditions occur:

- Out of space condition
- Maximum extents reached condition
- Space quota exceeded condition

For a nonresumable statement, these conditions result in errors and the statement is rolled back.

Suspending a statement automatically results in suspending the transaction. Thus all transactional resources are held through a statement suspend and resume.

When the error condition disappears (for example, as a result of user intervention or perhaps sort space released by other queries), the suspended statement automatically resumes execution.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about enabling resumable space allocation, what conditions are correctable, and what statements can be made resumable.

Oracle and Transaction Management

A transaction in Oracle begins when the first executable SQL statement is encountered. An **executable SQL statement** is a SQL statement that generates calls to an instance, including DML and DDL statements.

When a transaction begins, Oracle assigns the transaction to an available rollback segment to record the rollback entries for the new transaction.

A transaction ends when any of the following occurs:

- You issue a `COMMIT` or `ROLLBACK` statement without a `SAVEPOINT` clause.

- You execute a DDL statement such as `CREATE`, `DROP`, `RENAME`, or `ALTER`. If the current transaction contains any DML statements, Oracle first commits the transaction, and then executes and commits the DDL statement as a new, single statement transaction.
- A user disconnects from Oracle. The current transaction is committed.
- A user process terminates abnormally. The current transaction is rolled back.

After one transaction ends, the next executable SQL statement automatically starts the following transaction.

Note: Applications should always explicitly commit or roll back transactions before program termination.

See Also: "Transactions and Rollback Segments" on page 3-26

Commit Transactions

Committing a transaction means making permanent the changes performed by the SQL statements within the transaction.

Before a transaction that modifies data is committed, the following has occurred:

- Oracle has generated rollback segment records in rollback segment buffers of the system global area (SGA). The rollback information contains the old data values changed by the SQL statements of the transaction.
- Oracle has generated redo log entries in the redo log buffer of the SGA. The redo log record contains the change to the data block and the change to the rollback block. These changes may go to disk before a transaction is committed.
- The changes have been made to the database buffers of the SGA. These changes may go to disk before a transaction is committed.

Note: The data changes for a committed transaction, stored in the database buffers of the SGA, are not necessarily written immediately to the datafiles by the database writer (DBW*n*) background process. This writing takes place when it is most efficient for the database to do so. It can happen before the transaction commits or, alternatively, it can happen some time after the transaction commits.

When a transaction is committed, the following occurs:

1. The internal transaction table for the associated rollback segment records that the transaction has committed, and the corresponding unique system change number (SCN) of the transaction is assigned and recorded in the table.
2. The log writer process (LGWR) writes redo log entries in the SGA's redo log buffers to the online redo log file. It also writes the transaction's SCN to the online redo log file. This atomic event constitutes the commit of the transaction.
3. Oracle releases locks held on rows and tables.
4. Oracle marks the transaction complete.

See Also:

- "Overview of Locking Mechanisms" on page 22-3
- "Oracle Processes Overview" on page 9-5 for more information about the background processes LGWR and DBW*n* for changes to rollback segments

Rollback of Transactions

Rolling back means undoing any changes to data that have been performed by SQL statements within an uncommitted transaction. Oracle uses rollback segments to store old values. The redo log contains a record of changes.

Oracle allows you to roll back an entire uncommitted transaction. Alternatively, you can roll back the trailing portion of an uncommitted transaction to a marker called a savepoint.

All types of rollbacks use the same procedures:

- Statement-level rollback (due to statement or deadlock execution error)
- Rollback to a savepoint
- Rollback of a transaction due to user request
- Rollback of a transaction due to abnormal process termination
- Rollback of all outstanding transactions when an instance terminates abnormally
- Rollback of incomplete transactions during recovery

In rolling back **an entire transaction**, without referencing any savepoints, the following occurs:

1. Oracle undoes all changes made by all the SQL statements in the transaction by using the corresponding rollback segments.
2. Oracle releases all the transaction's locks of data.
3. The transaction ends.

See Also:

- "Savepoints In Transactions" on page 17-8
- "Overview of Locking Mechanisms" on page 22-3
- *Oracle9i Recovery Manager User's Guide and Reference* for information about what happens to committed and uncommitted changes during recovery

Savepoints In Transactions

You can declare intermediate markers called **savepoints** within the context of a transaction. Savepoints divide a long transaction into smaller parts.

Using savepoints, you can arbitrarily mark your work at any point within a long transaction. You then have the option later of rolling back work performed before the current point in the transaction but after a declared savepoint within the transaction. For example, you can use savepoints throughout a long complex series of updates, so if you make an error, you do not need to resubmit every statement.

Savepoints are similarly useful in application programs. If a procedure contains several functions, then you can create a savepoint before each function begins. Then, if a function fails, it is easy to return the data to its state before the function began and re-execute the function with revised parameters or perform a recovery action.

After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. Other transactions that were waiting for the previously locked resources can proceed. Other transactions that want to update previously locked rows can do so.

When a transaction is rolled back to a savepoint, the following occurs:

1. Oracle rolls back only the statements executed after the savepoint.
2. Oracle preserves the specified savepoint, but all savepoints that were established after the specified one are lost.
3. Oracle releases all table and row locks acquired since that savepoint but retains all data locks acquired previous to the savepoint.

The transaction remains active and can be continued.

Note: Whenever a session is waiting on a transaction, a rollback to savepoint does not free row locks. To make sure a transaction doesn't hang if it cannot obtain a lock, use

`FOR UPDATE ... NOWAIT`

before issuing `UPDATE` or `DELETE` statements.

Transaction Naming

Oracle9*i*, Release 1 (9.0.1), lets you name a transaction, using a simple and memorable text string. This name is a reminder of what the transaction is about. Transaction names replace commit comments for distributed transactions, with 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 DBA can view transaction names in the Oracle Enterprise Manager, when monitoring system activity.
- Transaction names are written to the transaction auditing redo record, if compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher.
- Log Miner can use transaction names to search for a specific transaction from transaction auditing records in the redo log.
- You can use transaction names to find a specific transaction in data dictionary tables, such as `V$TRANSACTION`.

How Transactions Are Named

You name a transaction by using the `SET TRANSACTION ... NAME` command before you start the transaction.

When you name a transaction, you associate the transaction's name with its ID. Transaction names do not have to be unique; different transactions can have the same transaction name at the same time by the same owner. You can use any name that enables you to distinguish the transaction.

Commit Comment

In previous releases, you could associate a comment with a transaction by using a commit comment. However, a comment can be associated with a transaction only when a transaction is being committed.

Commit comments are still supported for backward compatibility. However, we strongly recommend that you use transaction names. Commit comments are ignored in named transactions.

Note: In a future release, commit comments will be deprecated.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about distributed transactions
- *Oracle9i SQL Reference* for more information about transaction naming syntax

The Two-Phase Commit Mechanism

In a distributed database, Oracle must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

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.

A **two-phase commit** mechanism guarantees that *all* database servers participating in a distributed transaction either all commit or all roll back the statements in the transaction. A two-phase commit mechanism also protects implicit DML operations performed by integrity constraints, remote procedure calls, and triggers.

The Oracle two-phase commit mechanism is completely 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 to commit the transaction. No coding or complex statement syntax is required to include distributed transactions within the body of a database application.

The recoverer (`RECO`) background process automatically resolves the outcome of **in-doubt distributed transactions**—distributed transactions in which the commit was interrupted by any type of system or network failure. After the failure is repaired and communication is reestablished, the `RECO` process of each local

Oracle server automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

In the event of a long-term failure, Oracle allows each local administrator to manually commit or roll back any distributed transactions that are in doubt as a result of the failure. This option enables the local database administrator to free any locked resources that are held indefinitely as a result of the long-term failure.

If a database must be recovered to a point in the past, Oracle's recovery facilities enable database administrators at other sites to return their databases to the earlier point in time also. This operation ensures that the global database remains consistent.

See Also: *Oracle9i Heterogeneous Connectivity Administrator's Guide*

Discrete Transaction Management

Application developers can improve the performance of short, nondistributed transactions by using the `BEGIN_DISCRETE_TRANSACTION` procedure. This procedure streamlines transaction processing so that short transactions can execute more rapidly.

During a discrete transaction, all changes made to any data are deferred until the transaction commits. Of course, other concurrent transactions are unable to see the uncommitted changes of a transaction whether the transaction is discrete or not.

The following events occur during a discrete transaction:

1. Oracle generates redo information, but stores it in a separate location in memory.
2. When the transaction issues a commit request, Oracle writes the redo information to the redo log file along with other group commits.
3. Oracle applies the changes to the database block directly to the block.
4. Oracle returns control to the application after the commit completes.

This transaction design eliminates the need to generate undo information, because the block is not modified until the transaction is committed, and the redo information is stored in the redo log buffers.

There is no interaction between discrete transactions, which always generate redo, and the `NOLOGGING` mode, which applies only to direct path operations. Discrete transactions can therefore be issued against tables that have the `NOLOGGING` attribute set.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information about the `BEGIN_DISCRETE_TRANSACTION` procedure

Autonomous Transactions

Autonomous transactions are independent transactions that can be called from within another transaction. An autonomous transaction lets you leave the context of the calling transaction, perform some SQL operations, commit or roll back those operations, and then return to the calling transaction's context and continue with that transaction.

Once invoked, an autonomous transaction is totally independent of the main transaction that called it. It does not see any of the uncommitted changes made by the main transaction and does not share any locks or resources with the main transaction. Changes made by an autonomous transaction become visible to other transactions upon commit of the autonomous transactions.

One autonomous transaction can call another. There are no limits, other than resource limits, on how many levels of autonomous transactions can be called.

Deadlocks are possible between an autonomous transaction and its calling transaction. Oracle detects such deadlocks and returns an error. The application developer is responsible for avoiding deadlock situations.

Autonomous transactions are useful for implementing actions that need to be performed independently, regardless of whether the calling transaction commits or rolls back, such as transaction logging and retry counters.

Autonomous PL/SQL Blocks

You can call autonomous transactions from within a PL/SQL block. Use the pragma `AUTONOMOUS_TRANSACTION`. A **pragma** is a compiler directive. You can declare the following kinds of PL/SQL blocks to be autonomous:

- Stored procedure or function
- Local procedure or function
- Package
- Type method
- Top-level anonymous block

When an autonomous PL/SQL block is entered, the transaction context of the caller is suspended. This operation ensures that SQL operations performed in this block (or other blocks called from it) have no dependence or effect on the state of the caller's transaction context.

When an autonomous block invokes another autonomous block or itself, the called block does not share any transaction context with the calling block. However, when an autonomous block invokes a non-autonomous block (that is, one that is not declared to be autonomous), the called block inherits the transaction context of the calling autonomous block.

See Also: *PL/SQL User's Guide and Reference*

Transaction Control Statements in Autonomous Blocks

Transaction control statements in an autonomous PL/SQL block apply only to the currently active autonomous transaction. Examples of such statements are:

```
SET TRANSACTION
COMMIT
ROLLBACK
SAVEPOINT
ROLLBACK TO SAVEPOINT
```

Similarly, transaction control statements in the main transaction apply only to that transaction and not to any autonomous transaction that it calls. For example, rolling back the main transaction to a savepoint taken before the beginning of an autonomous transaction does not roll back the autonomous transaction.

See Also: *PL/SQL User's Guide and Reference*

18

Triggers

This chapter discusses triggers, which are procedures written in PL/SQL, Java, or C that execute (fire) implicitly whenever a table or view is modified or when some user actions or database system actions occur. You can write triggers that fire whenever one of the following operations occurs: DML statements on a particular schema object, DDL statements issued within a schema or database, user logon or logoff events, server errors, database startup, or instance shutdown.

This chapter includes:

- Introduction to Triggers
- Parts of a Trigger
- Types of Triggers
- Trigger Execution

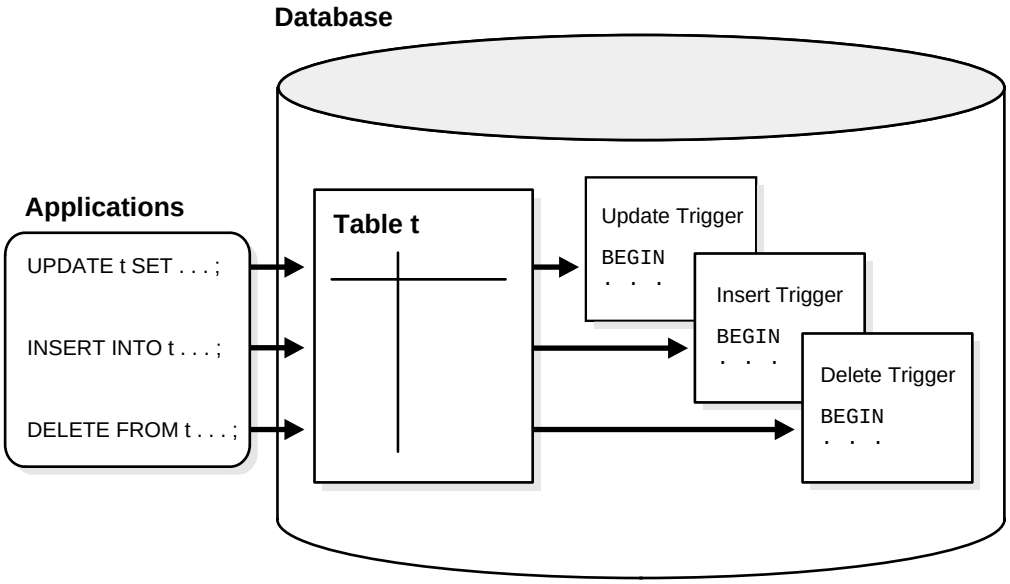
Introduction to Triggers

Oracle allows you to define procedures called **triggers** that execute implicitly when an `INSERT`, `UPDATE`, or `DELETE` statement is issued against the associated table or, in some cases, against a view, or when database system actions occur. These procedures can be written in PL/SQL or Java and stored in the database, or they can be written as C callouts.

Triggers are similar to stored procedures. A trigger stored in the database can include SQL and PL/SQL or Java statements to execute as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly executed by a user, application, or trigger. Triggers are implicitly fired by Oracle when a triggering event occurs, no matter which user is connected or which application is being used.

Figure 18–1 shows a database application with some SQL statements that implicitly fire several triggers stored in the database. Notice that the database stores triggers separately from their associated tables.

Figure 18–1 Triggers



A trigger can also call out to a C procedure, which is useful for computationally intensive operations.

The events that fire a trigger include the following:

- DML statements that modify data in a table (`INSERT`, `UPDATE`, `or DELETE`)
- DDL statements
- System events such as startup, shutdown, and error messages
- User events such as logon and logoff

Note: Oracle Forms can define, store, and execute triggers of a different sort. However, do not confuse Oracle Forms triggers with the triggers discussed in this chapter.

See Also:

- Chapter 16, "SQL, PL/SQL, and Java" for information on the similarities of triggers to stored procedures
- "The Triggering Event or Statement" on page 18-7

How Triggers Are Used

Triggers supplement the standard capabilities of Oracle to provide a highly customized database management system. For example, a trigger can restrict DML operations against a table to those issued during regular business hours. You can also use triggers to:

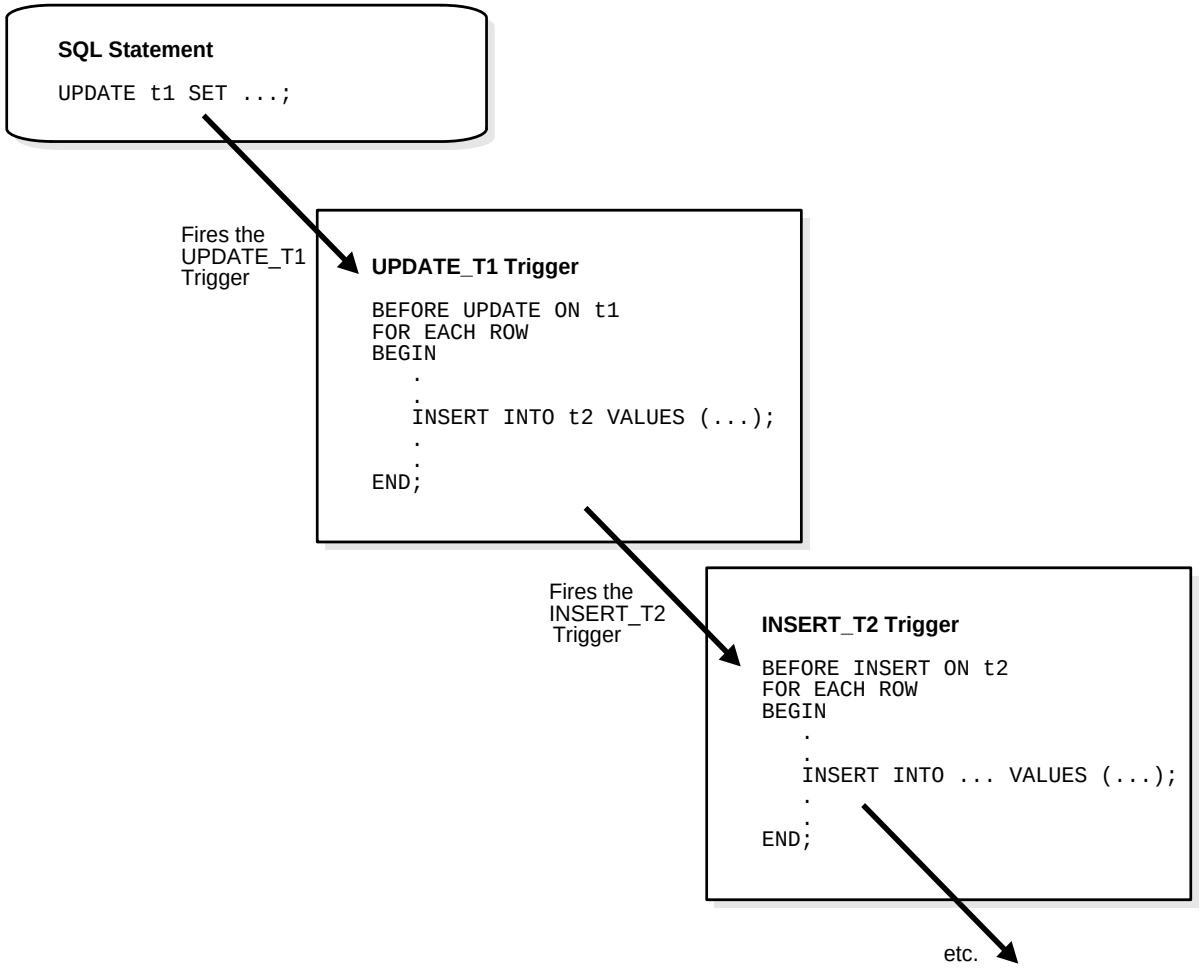
- Automatically generate derived column values
- Prevent invalid transactions
- Enforce complex security authorizations
- Enforce referential integrity across nodes in a distributed database
- Enforce complex business rules
- Provide transparent event logging
- Provide auditing
- Maintain synchronous table replicates
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Publish information about database events, user events, and SQL statements to subscribing applications

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for examples of trigger uses

Some Cautionary Notes about Triggers

Although triggers are useful for customizing a database, use them only when necessary. Excessive use of triggers can result in complex interdependencies, which can be difficult to maintain in a large application. For example, when a trigger fires, a SQL statement within its trigger action potentially can fire other triggers, resulting in **cascading triggers**. This can produce unintended effects. Figure 18–2 illustrates cascading triggers.

Figure 18–2 Cascading Triggers



Triggers compared with Declarative Integrity Constraints

You can use both triggers and integrity constraints to define and enforce any type of integrity rule. However, Oracle Corporation strongly recommends that you use triggers to constrain data input only in the following situations:

- To enforce referential integrity when child and parent tables are on different nodes of a distributed database
- To enforce complex business rules not definable using integrity constraints
- When a required referential integrity rule cannot be enforced using the following integrity constraints:
  - `NOT NULL, UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK`
  - `DELETE CASCADE`
  - `DELETE SET NULL`

  See Also: "How Oracle Enforces Data Integrity" on page 23-4 for more information about integrity constraints

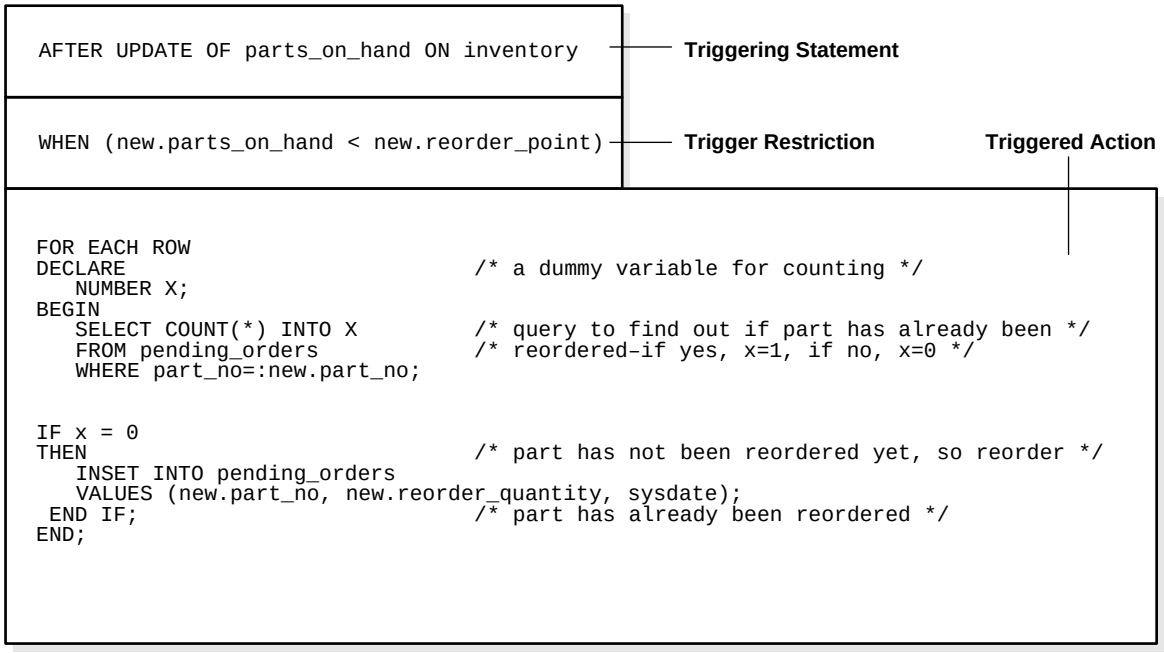
Parts of a Trigger

A trigger has three basic parts:

- A triggering event or statement
- A trigger restriction
- A trigger action

Figure 18–3 represents each of these parts of a trigger and is not meant to show exact syntax. The sections that follow explain each part of a trigger in greater detail.

Figure 18–3 The REORDER Trigger



The Triggering Event or Statement

A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:

- An `INSERT`, `UPDATE`, or `DELETE` statement on a specific table (or view, in some cases)
- A `CREATE`, `ALTER`, or `DROP` statement on any schema object
- A database startup or instance shutdown
- A specific error message or any error message
- A user logon or logoff

For example, in Figure 18–3, the triggering statement is:

```
... UPDATE OF parts_on_hand ON inventory ...
```

This statement means that when the PARTS_ON_HAND column of a row in the INVENTORY table is updated, fire the trigger. When the triggering event is an UPDATE statement, you can include a column list to identify which columns must be updated to fire the trigger. You cannot specify a column list for INSERT and DELETE statements, because they affect entire rows of information.

A triggering event can specify multiple SQL statements:

```
... INSERT OR UPDATE OR DELETE OF inventory ...
```

This part means that when an INSERT, UPDATE, or DELETE statement is issued against the INVENTORY table, fire the trigger. When multiple types of SQL statements can fire a trigger, you can use conditional predicates to detect the type of triggering statement. In this way, you can create a single trigger that executes different code based on the type of statement that fires the trigger.

Trigger Restriction

A trigger restriction specifies a Boolean expression that must be TRUE for the trigger to fire. The trigger action is not executed if the trigger restriction evaluates to FALSE or UNKNOWN. In the example, the trigger restriction is:

```
new.parts_on_hand < new.reorder_point
```

Consequently, the trigger does not fire unless the number of available parts is less than a present reorder amount.

Trigger Action

A trigger action is the procedure (PL/SQL block, Java program, or C callout) that contains the SQL statements and code to be executed when the following events occur:

- A triggering statement is issued.
- The trigger restriction evaluates to TRUE.

Like stored procedures, a trigger action can:

- Contain SQL, PL/SQL, or Java statements
- Define PL/SQL language constructs such as variables, constants, cursors, exceptions
- Define Java language constructs
- Call stored procedures

If the triggers are row triggers, the statements in a trigger action have access to column values of the row being processed by the trigger. Correlation names provide access to the old and new values for each column.

Types of Triggers

This section describes the different types of triggers:

- Row Triggers and Statement Triggers
- BEFORE and AFTER Triggers
- INSTEAD OF Triggers
- Triggers on System Events and User Events

Row Triggers and Statement Triggers

When you define a trigger, you can specify the number of times the trigger action is to be executed:

- Once for every row affected by the triggering statement, such as a trigger fired by an `UPDATE` statement that updates many rows
- Once for the triggering statement, no matter how many rows it affects

Row Triggers

A **row trigger** is fired each time the table is affected by the triggering statement. For example, if an `UPDATE` statement updates multiple rows of a table, a row trigger is fired once for each row affected by the `UPDATE` statement. If a triggering statement affects no rows, a row trigger is not executed.

Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected. For example, Figure 18–3 illustrates a row trigger that uses the values of each row affected by the triggering statement.

Statement Triggers

A **statement trigger** is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects, even if no rows are affected. For example, if a `DELETE` statement deletes several 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. For example, use a statement trigger to:

- Make a complex security check on the current time or user
- Generate a single audit record

BEFORE and AFTER Triggers

When defining a trigger, you can specify the **trigger timing**—whether the trigger action is to be executed before or after the triggering statement. `BEFORE` and `AFTER` apply to both statement and row triggers.

`BEFORE` and `AFTER` triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the base tables of a view are fired if an `INSERT`, `UPDATE`, or `DELETE` statement is issued against the view. `BEFORE` and `AFTER` triggers fired by DDL statements can be defined only on the database or a schema, not on particular tables.

See Also:

- "INSTEAD OF Triggers" on page 18-13
- "Triggers on System Events and User Events" on page 18-16 for information about how `BEFORE` and `AFTER` triggers can be used to publish information about DML and DDL statements

BEFORE Triggers

`BEFORE` triggers execute the trigger action before the triggering statement is executed. This type of trigger is commonly used in the following situations:

- When the trigger action determines whether the triggering statement should be allowed to complete. Using a `BEFORE` trigger for this purpose, you can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action.
- To derive specific column values before completing a triggering `INSERT` or `UPDATE` statement.

AFTER Triggers

`AFTER` triggers execute the trigger action after the triggering statement is executed.

Trigger Type Combinations

Using the options listed previously, you can create four types of row and statement triggers:

- **BEFORE** *statement* **trigger**

  Before executing the triggering statement, the trigger action is executed.

- **BEFORE** ***row*** **trigger**

  Before modifying each row affected by the triggering statement and before checking appropriate integrity constraints, the trigger action is executed, provided that the trigger restriction was not violated.

- **AFTER** ***row*** **trigger**

  After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row provided the trigger restriction was not violated. Unlike `BEFORE` *`row`* triggers, `AFTER` *`row`* triggers lock rows.

- **AFTER** ***statement*** **trigger**

  After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.

You can have multiple triggers of the same type for the same statement for any given table. For example, you can have two `BEFORE` *`statement`* triggers for `UPDATE` statements on the `EMP` table. Multiple triggers of the same type permit modular installation of applications that have triggers on the same tables. Also, Oracle materialized view logs use `AFTER` *`row`* triggers, so you can design your own `AFTER` *`row`* trigger in addition to the Oracle-defined `AFTER` *`row`* trigger.

You can create as many triggers of the preceding different types as you need for each type of DML statement, `(INSERT, UPDATE,` or `DELETE)`.

For example, suppose you have a table, `SAL,` and you want to know when the table is being accessed and the types of queries being issued. The following example contains a sample package and trigger that tracks this information by hour and type of action (for example, `UPDATE, DELETE,` or `INSERT)` on table `SAL.` The global session variable `STAT.ROWCNT` is initialized to zero by a `BEFORE` *`statement`* trigger. Then it is increased each time the row trigger is executed. Finally the statistical information is saved in the table `STAT_TAB` by the `AFTER` *`statement`* trigger.

Sample Package and Trigger for SAL Table

```
DROP TABLE stat_tab;
CREATE TABLE stat_tab(utype CHAR(8),
      rowcnt INTEGER, uhour INTEGER);

CREATE OR REPLACE PACKAGE stat IS
 rowcnt INTEGER;
END;
/

CREATE TRIGGER bt BEFORE UPDATE OR DELETE OR INSERT ON sal
BEGIN
 stat.rowcnt := 0;
END;
/

CREATE TRIGGER rt BEFORE UPDATE OR DELETE OR INSERT ON sal
FOR EACH ROW BEGIN
 stat.rowcnt := stat.rowcnt + 1;
END;
/

CREATE TRIGGER at AFTER UPDATE OR DELETE OR INSERT ON sal
DECLARE
 typ  CHAR(8);
 hour NUMBER;
BEGIN
 IF updating
 THEN typ := 'update'; END IF;
 IF deleting  THEN typ := 'delete'; END IF;
 IF inserting THEN typ := 'insert'; END IF;

 hour := TRUNC((SYSDATE - TRUNC(SYSDATE)) * 24);
 UPDATE stat_tab
    SET rowcnt = rowcnt + stat.rowcnt
   WHERE utype = typ
     AND uhour = hour;
 IF SQL%ROWCOUNT = 0 THEN
   INSERT INTO stat_tab VALUES (typ, stat.rowcnt, hour);
 END IF;

EXCEPTION
 WHEN dup_val_on_index THEN
   UPDATE stat_tab
      SET rowcnt = rowcnt + stat.rowcnt
```

```
     WHERE utype = typ
       AND uhour = hour;
END;
/
```

INSTEAD OF Triggers

INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.

You can write normal INSERT, UPDATE, and DELETE statements against the view and the INSTEAD OF trigger is fired to update the underlying tables appropriately. INSTEAD OF triggers are activated for each row of the view that gets modified.

Modify Views

Modifying views can have ambiguous results:

- Deleting a row in a view could either mean deleting it from the base table or updating some column values so that it is no longer selected by the view.
- Inserting a row in a view could either mean inserting a new row into the base table or updating an existing row so that it is projected by the view.
- Updating a column in a view that involves joins might change the semantics of other columns that are not projected by the view.

Object views present additional problems. For example, a key use of object views is to represent master/detail relationships. This operation inevitably involves joins, but modifying joins is inherently ambiguous.

As a result of these ambiguities, there are many restrictions on which views are modifiable. An INSTEAD OF trigger can be used on object views as well as relational views that are not otherwise modifiable.

Even if the view is inherently modifiable, you might want to perform validations on the values being inserted, updated or deleted. INSTEAD OF triggers can also be used in this case. Here the trigger code performs the validation on the rows being modified and if valid, propagate the changes to the underlying tables.

INSTEAD OF triggers also enable you to modify object view instances on the client-side through OCI. To modify an object materialized by an object view in the

client-side object cache and flush it back to the persistent store, you must specify `INSTEAD OF` triggers, unless the object view is inherently modifiable. However, it is not necessary to define these triggers for just pinning and reading the view object in the object cache.

See Also:

- Chapter 15, "Object Views"
- *Oracle Call Interface Programmer's Guide*

Views That Are Not Modifiable

A view is **inherently modifiable** if data can be inserted, updated, or deleted without using `INSTEAD OF` triggers and if it conforms to the restrictions listed as follows. If the view query contains any of the following constructs, the view is not inherently modifiable and you therefore cannot perform inserts, updates, or deletes on the view:

- Set operators
- Aggregate functions
- `GROUP BY`, `CONNECT BY`, or `START WITH` clauses
- The `DISTINCT` operator
- Joins (however, some join views are updatable)

If a view contains pseudocolumns or expressions, you can only update the view with an `UPDATE` statement that does not refer to any of the pseudocolumns or expressions.

See Also: "Updatable Join Views" on page 11-20

Example of an INSTEAD OF Trigger

The following example shows an `INSTEAD OF` trigger for updating rows in the `manager_info` view, which lists all the departments and their managers.

Let `dept` be a relational table containing a list of departments:

```
CREATE TABLE dept (
   deptno NUMBER PRIMARY KEY,
   deptname VARCHAR2(20),
   manager_num NUMBER
);
```

Let `emp` be a relational table containing the list of employees and the departments in which they work.

```
CREATE TABLE emp (
   empno NUMBER PRIMARY KEY,
   empname VARCHAR2(20),
   deptno NUMBER REFERENCES dept(deptno),
   startdate DATE
);
ALTER TABLE dept ADD (FOREIGN KEY(manager_num) REFERENCES emp(empno));
```

Create the `manager_info` view that lists all the managers for each department:

```
CREATE VIEW manager_info AS
  SELECT d.deptno, d.deptname, e.empno, e.empname
  FROM   emp e, dept d
  WHERE  e.empno =  d.manager_num;
```

Now, define an `INSTEAD OF` trigger to handle the inserts on the view. An insert into the `manager_info` view can be translated into an update to the `manager_num` column of the `dept` table.

In the trigger, you can also enforce the constraint that there must be at least one employee working in the department for a person to be a manager of that department.

```
CREATE TRIGGER manager_info_insert
  INSTEAD OF INSERT ON manager_info
    REFERENCING NEW AS n              -- new manager information
  FOR EACH ROW
  DECLARE
    empCount NUMBER;
  BEGIN

     /* First check to make sure that the number of employees
      * in the department is greater than one */
     SELECT COUNT(*) INTO empCount
      FROM emp e
      WHERE e.deptno = :n.deptno;

     /* If there are enough employees then make him or her the manager */
     IF empCount >= 1 THEN

        UPDATE dept d
         SET manager_num = :n.empno
         WHERE d.deptno = :n.deptno;
```

```
    END IF;
END;
/
```

Any inserts to the `manager_info` view, such as:

```
INSERT INTO manager_info VALUES (200,'Sports',1002,'Jack');
```

fires the `manager_info_insert` trigger and updates the underlying tables. Similar triggers can specify appropriate actions for `INSERT` and `DELETE` on the view.

INSTEAD OF Triggers on Nested Tables

You cannot modify the elements of a nested table column in a view directly with the `TABLE` clause. However, you can do so by defining an `INSTEAD OF` trigger on the nested table column of the view. The triggers on the nested tables fire if a nested table element is updated, inserted, or deleted and handle the actual modifications to the underlying tables.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals*
- `CREATE TRIGGER` statement in *Oracle9i SQL Reference*

Triggers on System Events and User Events

You can use triggers to publish information about database events to subscribers. Applications can subscribe to database events just as they subscribe to messages from other applications. These database events can include:

- System events
  - Database startup and shutdown
  - Server error message events
- User events
  - User logon and logoff
  - DDL statements (`CREATE`, `ALTER`, and `DROP`)
  - DML statements (`INSERT`, `DELETE`, and `UPDATE`)

Triggers on system events can be defined at the database level or schema level. For example, a database shutdown trigger is defined at the database level:

```
CREATE TRIGGER register_shutdown
  ON DATABASE
  SHUTDOWN
    BEGIN
    ...
    DBMS_AQ.ENQUEUE(...);
    ...
    END;
```

Triggers on DDL statements or logon/logoff events can also be defined at the database level or schema level. Triggers on DML statements can be defined on a table or view. A trigger defined at the database level fires for all users, and a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Event Publication

Event publication uses the publish-subscribe mechanism of Oracle Advanced Queuing. A **queue** serves as a message repository for subjects of interest to various subscribers. Triggers use the `DBMS_AQ` package to enqueue a message when specific system or user events occur.

See Also:

- *Oracle9i Application Developer's Guide - Advanced Queuing*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

Event Attributes

Each event allows the use of attributes within the trigger text. For example, the database startup and shutdown triggers have attributes for the instance number and the database name, and the logon and logoff triggers have attributes for the username. You can specify a function with the same name as an attribute when you create a trigger if you want to publish that attribute when the event occurs. The attribute's value is then passed to the function or payload when the trigger fires. For triggers on DML statements, the `:OLD` column values pass the attribute's value to the `:NEW` column value.

System Events

System events that can fire triggers are related to instance startup and shutdown and error messages. Triggers created on startup and shutdown events have to be associated with the database. Triggers created on error events can be associated with the database or with a schema.

- `STARTUP` triggers fire when the database is opened by an instance. Their attributes include the system event, instance number, and database name.
- `SHUTDOWN` triggers fire just before the server starts shutting down an instance. You can use these triggers to make subscribing applications shut down completely when the database shuts down. For abnormal instance shutdown, these triggers cannot be fired. The attributes of `SHUTDOWN` triggers include the system event, instance number, and database name.
- `SERVERERROR` triggers fire when a specified error occurs, or when any error occurs if no error number is specified. Their attributes include the system event and error number.

User Events

User events that can fire triggers are related to user logon and logoff, DDL statements, and DML statements.

Triggers on LOGON and LOGOFF Events `LOGON` and `LOGOFF` triggers can be associated with the database or with a schema. Their attributes include the system event and username, and they can specify simple conditions on `USERID` and `USERNAME.`

- `LOGON` triggers fire after a successful logon of a user.
- `LOGOFF` triggers fire at the start of a user logoff.

Triggers on DDL Statements DDL triggers can be associated with the database or with a schema. Their attributes include the system event, the type of schema object, and its name. They can specify simple conditions on the type and name of the schema object, as well as functions like `USERID` and `USERNAME.` DDL triggers include the following types of triggers:

- `BEFORE CREATE` and `AFTER CREATE` triggers fire when a schema object is created in the database or schema.
- `BEFORE ALTER` and `AFTER ALTER` triggers fire when a schema object is altered in the database or schema.

- BEFORE DROP and AFTER DROP triggers fire when a schema object is dropped from the database or schema.

Triggers on DML Statements DML triggers for event publication are associated with a table. They can be either BEFORE or AFTER triggers that fire for each row on which the specified DML operation occurs. You cannot use INSTEAD OF triggers on views to publish events related to DML statements—instead, you can publish events using BEFORE or AFTER triggers for the DML operations on a view's underlying tables that are caused by INSTEAD OF triggers.

The attributes of DML triggers for event publication include the system event and the columns defined by the user in the SELECT list. They can specify simple conditions on the type and name of the schema object, as well as functions (such as UID, USER, USERENV, and SYSDATE), pseudocolumns, and columns. The columns can be prefixed by :OLD and :NEW for old and new values. Triggers on DML statements include the following triggers:

- BEFORE INSERT and AFTER INSERT triggers fire for each row inserted into the table.
- BEFORE UPDATE and AFTER UPDATE triggers fire for each row updated in the table.
- BEFORE DELETE and AFTER DELETE triggers fire for each row deleted from the table.

See Also:

- "Row Triggers" on page 18-9
- "BEFORE and AFTER Triggers" on page 18-10
- *Oracle9i Application Developer's Guide - Fundamentals* for more information about event publication using triggers on system events and user events

Trigger Execution

A trigger is in either of two distinct modes:

Enabled — An enabled trigger executes its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE.

Disabled	A disabled trigger does not execute its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to `TRUE`.

For enabled triggers, Oracle automatically performs the following actions:

- Executes triggers of each type in a planned firing sequence when more than one trigger is fired by a single SQL statement
- Performs integrity constraint checking at a set point in time with respect to the different types of triggers and guarantees that triggers cannot compromise integrity constraints
- Provides read-consistent views for queries and constraints
- Manages the dependencies among triggers and schema objects referenced in the code of the trigger action
- Uses two-phase commit if a trigger updates remote tables in a distributed database
- Fires multiple triggers in an unspecified order, if more than one trigger of the same type exists for a given statement

The Execution Model for Triggers and Integrity Constraint Checking

A single SQL statement can potentially fire up to four types of triggers:

- `BEFORE` *`row`* triggers
- `BEFORE` *`statement`* triggers
- `AFTER` *`row`* triggers
- `AFTER` *`statement`* triggers

A triggering statement or a statement within a trigger can cause one or more integrity constraints to be checked. Also, triggers can contain statements that cause other triggers to fire (cascading triggers).

Oracle uses the following execution model to maintain the proper firing sequence of multiple triggers and constraint checking:

1. Execute all `BEFORE` *`statement`* triggers that apply to the statement.
2. Loop for each row affected by the SQL statement.
   a. Execute all `BEFORE` *`row`* triggers that apply to the statement.

b. Lock and change row, and perform integrity constraint checking. (The lock is not released until the transaction is committed.)

c. Execute all AFTER *row* triggers that apply to the statement.

3. Complete deferred integrity constraint checking.

4. Execute all AFTER *statement* triggers that apply to the statement.

The definition of the execution model is recursive. For example, a given SQL statement can cause a BEFORE *row* trigger to be fired and an integrity constraint to be checked. That BEFORE *row* trigger, in turn, might perform an update that causes an integrity constraint to be checked and an AFTER *statement* trigger to be fired. The AFTER *statement* trigger causes an integrity constraint to be checked. In this case, the execution model executes the steps recursively, as follows:

Original SQL statement issued.

1. BEFORE *row* triggers fired.

a. AFTER *statement* triggers fired by UPDATE in BEFORE *row* trigger.

i. Statements of AFTER *statement* triggers executed.

ii. Integrity constraint checked on tables changed by AFTER *statement* triggers.

b. Statements of BEFORE *row* triggers executed.

c. Integrity constraint checked on tables changed by BEFORE *row* triggers.

2. SQL statement executed.

3. Integrity constraint from SQL statement checked.

There are two exceptions to this recursion:

- When a triggering statement modifies one table in a referential constraint (either the primary key or foreign key table), and a triggered statement modifies the other, only the triggering statement will check the integrity constraint. This allows row triggers to enhance referential integrity.
- Statement triggers fired due to DELETE CASCADE and DELETE SET NULL are fired before and after the user DELETE statement, not before and after the individual enforcement statements. This prevents those statement triggers from encountering mutating errors.

An important property of the execution model is that all actions and checks done as a result of a SQL statement must succeed. If an exception is raised within a trigger, and the exception is not explicitly handled, all actions performed as a result of the

original SQL statement, including the actions performed by fired triggers, are rolled back. Thus, integrity constraints cannot be compromised by triggers. The execution model takes into account integrity constraints and disallows triggers that violate declarative integrity constraints.

For example, in the previously outlined scenario, suppose that Steps 1 through 8 succeed; however, in Step 9 the integrity constraint is violated. As a result of this violation, all changes made by the SQL statement (in Step 8), the fired BEFORE *row* trigger (in Step 6), and the fired AFTER *statement* trigger (in Step 4) are rolled back.

Note: Although triggers of different types are fired in a specific order, triggers of the same type for the same statement are not guaranteed to fire in any specific order. For example, all BEFORE *row* triggers for a single UPDATE statement may not always fire in the same order. Design your applications so they do not rely on the firing order of multiple triggers of the same type.

Data Access for Triggers

When a trigger is fired, the tables referenced in the trigger action might be currently undergoing changes by SQL statements in other users' transactions. In all cases, the SQL statements executed within triggers follow the common rules used for standalone SQL statements. In particular, if an uncommitted transaction has modified values that a trigger being fired either needs to read (query) or write (update), the SQL statements in the body of the trigger being fired use the following guidelines:

- Queries see the current read-consistent materialized view of referenced tables and any data changed within the same transaction.
- Updates wait for existing data locks to be released before proceeding.

The following examples illustrate these points.

Data Access for Triggers Example 1 Assume that the SALARY_CHECK trigger (body) includes the following SELECT statement:

```
SELECT minsal, maxsal INTO minsal, maxsal
  FROM salgrade
  WHERE job_classification = :new.job_classification;
```

For this example, assume that transaction T1 includes an update to the MAXSAL column of the SALGRADE table. At this point, the SALARY_CHECK trigger is fired by a statement in transaction T2. The SELECT statement within the fired trigger (originating from T2) does not see the update by the uncommitted transaction T1, and the query in the trigger returns the old MAXSAL value as of the read-consistent point for transaction T2.

Data Access for Triggers Example 2 Assume that the TOTAL_SALARY trigger maintains a derived column that stores the total salary of all members in a department:

```
CREATE TRIGGER total_salary
AFTER DELETE OR INSERT OR UPDATE OF deptno, sal ON emp
 FOR EACH ROW BEGIN
  /* assume that DEPTNO and SAL are non-null fields */
   IF DELETING OR (UPDATING AND :old.deptno != :new.deptno)
   THEN UPDATE dept
  SET total_sal = total_sal - :old.sal
  WHERE deptno = :old.deptno;
   END IF;
   IF INSERTING OR (UPDATING AND :old.deptno != :new.deptno)
   THEN UPDATE dept
    SET total_sal = total_sal + :new.sal
    WHERE deptno = :new.deptno;
   END IF;
   IF (UPDATING AND :old.deptno = :new.deptno AND
    :old.sal != :new.sal )
   THEN UPDATE dept
    SET total_sal = total_sal - :old.sal + :new.sal
   WHERE deptno = :new.deptno;
  END IF;
 END;
```

For this example, suppose that one user's uncommitted transaction includes an update to the TOTAL_SAL column of a row in the DEPT table. At this point, the TOTAL_SALARY trigger is fired by a second user's SQL statement. Because the **uncommitted** transaction of the first user contains an update to a pertinent value in the TOTAL_SAL column (that is, a row lock is being held), the updates performed by the TOTAL_SALARY trigger are not executed until the transaction holding the row lock is committed or rolled back. Therefore, the second user waits until the commit or rollback point of the first user's transaction.

Storage of PL/SQL Triggers

Oracle stores PL/SQL triggers in compiled form, just like stored procedures. When a `CREATE TRIGGER` statement commits, the compiled PL/SQL code, called P code (for pseudocode), is stored in the database and the source code of the trigger is flushed from the shared pool.

> **See Also:** *PL/SQL User's Guide and Reference* for more information about compiling and storing PL/SQL code

Execution of Triggers

Oracle executes a trigger internally using the same steps used for procedure execution. The only subtle difference is that a user has the right to fire a trigger if he or she has the privilege to execute the triggering statement. Other than this, triggers are validated and executed the same way as stored procedures.

> **See Also:** *PL/SQL User's Guide and Reference* for more information about stored procedures

Dependency Maintenance for Triggers

Like procedures, triggers depend on referenced objects. Oracle automatically manages the dependencies of a trigger on the schema objects referenced in its trigger action. The dependency issues for triggers are the same as those for stored procedures. Triggers are treated like stored procedures. They are inserted into the data dictionary.

> **See Also:** Chapter 19, "Dependencies Among Schema Objects"

19

Dependencies Among Schema Objects

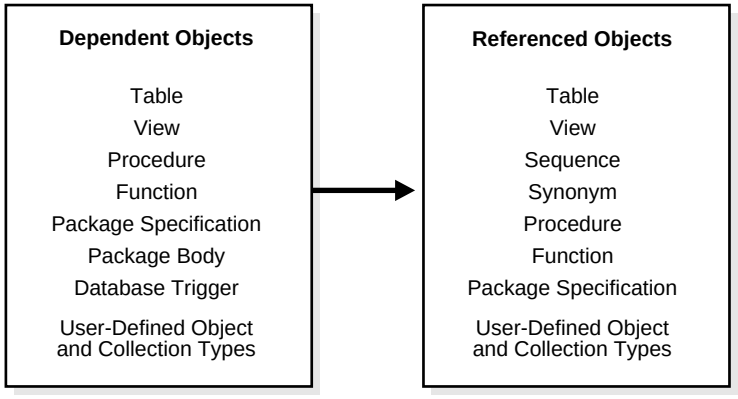
The definitions of some objects, including views and procedures, reference other objects, such as tables. As a result, the objects being defined are dependent on the objects referenced in their definitions. This chapter discusses the dependencies among schema objects and how Oracle automatically tracks and manages these dependencies. It includes:

- Introduction to Dependency Issues
- Resolution of Schema Object Dependencies
- Dependency Management and Nonexistent Schema Objects
- Shared SQL Dependency Management
- Local and Remote Dependency Management

Introduction to Dependency Issues

Some types of schema objects can reference other objects as part of their definition. For example, a view is defined by a query that references tables or other views. A procedure's body can include SQL statements that reference other objects of a database. An object that references another object as part of its definition is called a *dependent* object, while the object being referenced is a **referenced** object. Figure 19–1 illustrates the different types of dependent and referenced objects:

Figure 19–1 Types of Possible Dependent and Referenced Schema Objects



If you alter the definition of a referenced object, dependent objects may or may not continue to function without error, depending on the type of alteration. For example, if you drop a table, no view based on the dropped table is usable.

Oracle automatically records dependencies among objects to alleviate the complex job of dependency management for the database administrator and users. For example, if you alter a table on which several stored procedures depend, Oracle automatically recompiles the dependent procedures the next time the procedures are referenced (executed or compiled against).

To manage dependencies among schema objects, all of the schema objects in a database have a status:

`VALID`	The schema object has been compiled and can be immediately used when referenced.

`INVALID` The schema object must be compiled before it can be used.

- In the case of procedures, functions, and packages, this means compiling the schema object.
- In the case of views, this means that the view must be reparsed, using the current definition in the data dictionary.

Only dependent objects can be invalid. Tables, sequences, and synonyms are always valid.

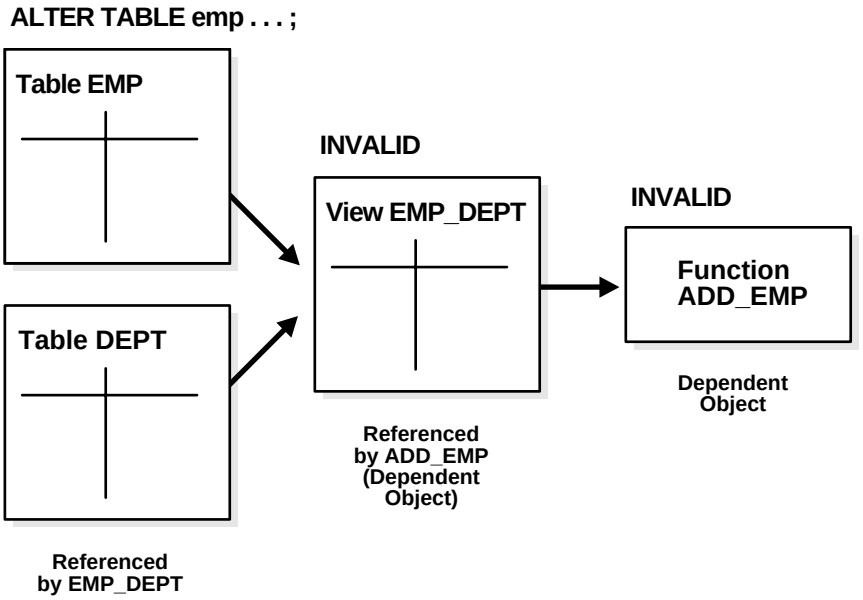
If a view, procedure, function, or package is invalid, Oracle may have attempted to compile it, but errors relating to the object occurred. For example, when compiling a view, one of its base tables might not exist, or the correct privileges for the base table might not be present. When compiling a package, there might be a PL/SQL or SQL syntax error, or the correct privileges for a referenced object might not be present. Schema objects with such problems remain invalid.

Oracle automatically tracks specific changes in the database and records the appropriate status for related objects in the data dictionary.

Status recording is a recursive process. Any change in the status of a referenced object changes the status not only for directly dependent objects, but also for indirectly dependent objects.

For example, consider a stored procedure that directly references a view. In effect, the stored procedure indirectly references the base tables of that view. Therefore, if you alter a base table, the view is invalidated, which then invalidates the stored procedure. Figure 19–2 illustrates indirect dependencies:

Figure 19–2 Indirect Dependencies



Resolution of Schema Object Dependencies

When a schema object is referenced directly in a SQL statement or indirectly through a reference to a dependent object, Oracle checks the status of the object explicitly specified in the SQL statement and any referenced objects, as necessary. Oracle's action depends on the status of the objects that are directly and indirectly referenced in a SQL statement:

- If every referenced object is valid, then Oracle executes the SQL statement immediately without any additional work.
- If any referenced view or procedure (including a function or package) is invalid, then Oracle automatically attempts to compile the object.
  - If all invalid referenced objects can be compiled successfully, then they are compiled and Oracle executes the SQL statement.

– If an invalid object cannot be compiled successfully, then it remains invalid. Oracle returns an error and rolls back the transaction containing the SQL statement.

Note: Oracle attempts to recompile an invalid object dynamically only if it has not been replaced since it was detected as invalid. This optimization eliminates unnecessary recompilations.

Compilation of Views and PL/SQL Program Units

A view or PL/SQL program unit can be compiled and made valid if the following conditions are satisfied:

- The definition of the view or program unit must be correct. All of the SQL and PL/SQL statements must be proper constructs.
- All referenced objects must be present and of the expected structure. For example, if the defining query of a view includes a column, the column must be present in the base table.
- The owner of the view or program unit must have the necessary privileges for the referenced objects. For example, if a SQL statement in a procedure inserts a row into a table, the owner of the procedure must have the `INSERT` privilege for the referenced table.

Views and Base Tables

A view depends on the base tables or views referenced in its defining query. If the defining query of a view is not explicit about which columns are referenced, for example, `SELECT * FROM table,` then the defining query is expanded when stored in the data dictionary to include all columns in the referenced base table at that time.

If a base table or view of a view is altered, renamed, or dropped, then the view is invalidated, but its definition remains in the data dictionary along with the privileges, synonyms, other objects, and other views that reference the invalid view.

> **Note:** Whenever you create a table, index, and view, and then drop the index, all objects dependent on that table are invalidated, including views, packages, package bodies, functions, and procedures.

An attempt to use an invalid view automatically causes Oracle to recompile the view dynamically. After replacing the view, the view might be valid or invalid, depending on the following conditions:

- All base tables referenced by the defining query of a view must exist. If a base table of a view is renamed or dropped, the view is invalidated and cannot be used. References to invalid views cause the referencing statement to fail. The view can be compiled only if the base table is renamed to its original name or the base table is re-created.
- If a base table is altered or re-created with the same columns, but the datatype of one or more columns in the base table is changed, then any dependent view can be recompiled successfully.
- If a base table of a view is altered or re-created with at least the same set of columns, then the view can be validated. The view cannot be validated if the base table is re-created with new columns and the view references columns no longer contained in the re-created table. The latter point is especially relevant in the case of views defined with a `SELECT * FROM table` query, because the defining query is expanded at view creation time and permanently stored in the data dictionary.

Program Units and Referenced Objects

Oracle automatically invalidates a program unit when the definition of a referenced object is altered. For example, assume that a standalone procedure includes several statements that reference a table, a view, another standalone procedure, and a public package procedure. In that case, the following conditions hold:

- If the referenced table is altered, then the dependent procedure is invalidated.
- If the base table of the referenced view is altered, then the view and the dependent procedure are invalidated.
- If the referenced standalone procedure is replaced, then the dependent procedure is invalidated.
- If the *body* of the referenced package is replaced, then the dependent procedure is not affected. However, if the **specification** of the referenced package is

replaced, then the dependent procedure is invalidated. This is a mechanism for minimizing dependencies among procedures and referenced objects by using packages.

- Whenever you create a table, index, and view, and then drop the index, all objects dependent on that table are invalidated, including views, packages, package bodies, functions, and procedures.

Data Warehousing Considerations

Some data warehouses drop indexes on tables at night to facilitate faster loads. However, all views dependent on the table whose index is dropped get invalidated. This means that subsequently running any package that reference these dropped views will invalidate the package.

Remember that whenever you create a table, index, and view, and then drop the index, all objects dependent on that table are invalidated, including views, packages, package bodies, functions, and procedures. This protects updatable join views.

To make the view valid again, use one of the following commands:

```
select * from vtest;
```

Or

```
alter view vtest compile
```

Session State and Referenced Packages

Each session that references a package construct has its own instance of that package, including a persistent state of any public and private variables, cursors, and constants. All of a session's package instantiations including state can be lost if any of the session's instantiated packages are subsequently invalidated and recompiled.

Security Authorizations

Oracle notices when a DML object or system privilege is granted to or revoked from a user or `PUBLIC` and automatically invalidates all the owner's dependent objects. Oracle invalidates the dependent objects to verify that an owner of a dependent object continues to have the necessary privileges for all referenced objects. Internally, Oracle notes that such objects do not have to be recompiled. Only security authorizations need to be validated, not the structure of any objects. This

optimization eliminates unnecessary recompilations and prevents the need to change a dependent object's time stamp.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information about forcing the recompilation of an invalid view or program unit

Function-Based Index Dependencies

Function-based indexes depend on functions used in the expression that defines the index. If a PL/SQL function or package function is changed, then the index is marked as disabled.

This section discusses requirements for function-based indexes and what happens when a function is changed in any manner, such as when it is dropped or privileges to use it are revoked.

Requirements

To create a function-based index:

- The following initialization parameters must be defined:
  - `QUERY_REWRITE_INTEGRITY` must be set to `TRUSTED`
  - `QUERY_REWRITE_ENABLED` must be set to `TRUE`
  - `COMPATIBLE` must set to `8.1.0.0.0` or a greater value
- The user must be granted `CREATE INDEX` and `QUERY REWRITE`, or `CREATE ANY INDEX` and `GLOBAL QUERY REWRITE`.

To use a function-based index:

- The table must be analyzed after the index is created.
- The query must be guaranteed not to need any `NULL` values from the indexed expression, because `NULL` values are not stored in indexes.

The following sections describe additional requirements.

> **See Also:** "Function-Based Indexes" on page 11-31

DETERMINISTIC Functions

Any user-written function used in a function-based index must have been declared with the `DETERMINISTIC` keyword to indicate that the function will always

return the same output return value for any given set of input argument values, now and in the future.

See Also: *Oracle9i Database Performance Guide and Reference*

Privileges on the Defining Function

The index owner needs the `EXECUTE` privilege on the function used to define a function-based index. If the `EXECUTE` privilege is revoked, Oracle marks the index `DISABLED`. The index owner does not need the `EXECUTE WITH GRANT OPTION` privilege on this function to grant `SELECT` privileges on the underlying table.

Resolve Dependencies of Function-Based Indexes

A function-based index depends on any function that it is using. If the function or the specification of a package containing the function is redefined (or if the index owner's `EXECUTE` privilege is revoked), then the following conditions hold:

- The index is marked as `DISABLED`.
- Queries on a `DISABLED` index fail if the optimizer chooses to use the index.
- DML operations on a `DISABLED` index fail unless the index is also marked `UNUSABLE` and the initialization parameter `SKIP_UNUSABLE_INDEXES` is set to true.

To re-enable the index after a change to the function, use the `ALTER INDEX ... ENABLE` statement.

Dependency Management and Nonexistent Schema Objects

When a dependent object is created, Oracle takes the following steps:

1. Oracle attempts to resolve all references by first searching in the current schema.
2. If a referenced object is not found in the current schema, Oracle attempts to resolve the reference by searching for a private synonym in the same schema.
3. If a private synonym is not found, Oracle looks for a public synonym.
4. If a public synonym is not found, Oracle searches for a schema name that matches the first portion of the object name.
5. If a matching schema name is found, Oracle attempts to find the object in that schema.

6. If no schema is found, an error is returned.

Because of how Oracle resolves references, it is possible for an object to depend on the **nonexistence** of other objects. This situation occurs when the dependent object uses a reference that would be interpreted differently were another object present. For example, assume the following:

- At the current point in time, the COMPANY schema contains a table named EMP.
- A PUBLIC synonym named EMP is created for COMPANY.EMP and the SELECT privilege for COMPANY.EMP is granted to the PUBLIC role.
- The JWARD schema does not contain a table or private synonym named EMP.
- The user JWARD creates a view in his schema with the following statement:

```
CREATE VIEW dept_salaries AS
   SELECT deptno, MIN(sal), AVG(sal), MAX(sal) FROM emp
   GROUP BY deptno
   ORDER BY deptno;
```

When JWARD creates the DEPT_SALARIES view, the reference to EMP is resolved by first looking for JWARD.EMP as a table, view, or private synonym, none of which is found, and then as a public synonym named EMP, which is found. As a result, Oracle notes that JWARD.DEPT_SALARIES depends on the nonexistence of JWARD.EMP and on the existence of PUBLIC.EMP.

Now assume that JWARD decides to create a new view named EMP in his schema using the following statement:

```
CREATE VIEW emp AS
   SELECT empno, ename, mgr, deptno
   FROM company.emp;
```

Notice that JWARD.EMP does not have the same structure as COMPANY.EMP.

As it attempts to resolve references in object definitions, Oracle internally makes note of dependencies that the new dependent object has on "nonexistent" objects—schema objects that, if they existed, would change the interpretation of the object's definition. Such dependencies must be noted in case a nonexistent object is later created. If a nonexistent object is created, all dependent objects must be invalidated so that dependent objects can be recompiled and verified and all dependent function-based indexes must be marked unusable.

Therefore, in the previous example, as JWARD.EMP is created, JWARD.DEPT_SALARIES is invalidated because it depends on JWARD.EMP. Then when

JWARD.DEPT_SALARIES is used, Oracle attempts to recompile the view. As Oracle resolves the reference to EMP, it finds JWARD.EMP (PUBLIC.EMP is no longer the referenced object). Because JWARD.EMP does not have a SAL column, Oracle finds errors when replacing the view, leaving it invalid.

In summary, you must manage dependencies on nonexistent objects checked during object resolution in case the nonexistent object is later created.

Shared SQL Dependency Management

In addition to managing dependencies among schema objects, Oracle also manages dependencies of each shared SQL area in the shared pool. If a table, view, synonym, or sequence is created, altered, or dropped, or a procedure or package specification is recompiled, all dependent shared SQL areas are invalidated. At a subsequent execution of the cursor that corresponds to an invalidated shared SQL area, Oracle reparses the SQL statement to regenerate the shared SQL area.

Local and Remote Dependency Management

Tracking dependencies and completing necessary recompilations are performed automatically by Oracle. **Local dependency management** occurs when Oracle manages dependencies among the objects in a single database. For example, a statement in a procedure can reference a table in the same database.

Remote dependency management occurs when Oracle manages dependencies in distributed environments across a network. For example, an Oracle Forms trigger can depend on a schema object in the database. In a distributed database, a local view's defining query can reference a remote table.

Management of Local Dependencies

Oracle manages all local dependencies using the database's internal dependency table, which keeps track of each schema object's dependent objects. When a referenced object is modified, Oracle uses the depends-on table to identify dependent objects, which are then invalidated.

For example, assume a stored procedure UPDATE_SAL references the table JWARD.EMP. If the definition of the table is altered in any way, the status of every object that references JWARD.EMP is changed to INVALID, including the stored procedure UPDATE_SAL. As a result, the procedure cannot be executed until it has been recompiled and is valid. Similarly, when a DML privilege is revoked from a user, every dependent object in the user's schema is invalidated. However, an

object that is invalid because authorization was revoked can be revalidated by "reauthorization," in which case it does not require full recompilation.

Management of Remote Dependencies

Oracle also manages application-to-database and distributed database dependencies. For example, an Oracle Forms application might contain a trigger that references a table, or a local stored procedure might call a remote procedure in a distributed database system. The database system must account for dependencies among such objects. Oracle uses different mechanisms to manage remote dependencies, depending on the objects involved.

Dependencies Among Local and Remote Database Procedures

Dependencies among stored procedures including functions, packages, and triggers in a distributed database system are managed using **time stamp checking** or **signature checking**.

The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` determines whether time stamps or signatures govern remote dependencies.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for details about managing remote dependencies with time stamps or signatures

Time stamp Checking In the time stamp checking dependency model, whenever a procedure is compiled or recompiled its **time stamp** (the time it is created, altered, or replaced) is recorded in the data dictionary. The time stamp is a record of the time the procedure is created, altered, or replaced. Additionally, the compiled version of the procedure contains information about each remote procedure that it references, including the remote procedure's schema, package name, procedure name, and time stamp.

When a dependent procedure is used, Oracle compares the remote time stamps recorded at compile time with the current time stamps of the remotely referenced procedures. Depending on the result of this comparison, two situations can occur:

- The local and remote procedures execute without compilation if the time stamps match.
- The local procedure is invalidated if any time stamps of remotely referenced procedures do not match, and an error is returned to the calling environment. Furthermore, all other local procedures that depend on the remote procedure with the new time stamp are also invalidated. For example, assume several

local procedures call a remote procedure, and the remote procedure is recompiled. When one of the local procedures is executed and notices the different time stamp of the remote procedure, every local procedure that depends on the remote procedure is invalidated.

Actual time stamp comparison occurs when a statement in the body of a local procedure executes a remote procedure. Only at this moment are the time stamps compared using the distributed database's communications link. Therefore, all statements in a local procedure that precede an invalid procedure call might execute successfully. Statements subsequent to an invalid procedure call do not execute at all. Compilation is required. However, any DML statements executed before the invalid procedure call are rolled back.

Signature Checking Oracle provides the additional capability of remote dependencies using **signatures**. The signature capability affects only remote dependencies. Local dependencies are not affected, as recompilation is always possible in this environment.

The signature of a procedure contains information about the following items:

- Name of the package, procedure, or function
- Base types of the parameters
- Modes of the parameters `(IN, OUT, and IN OUT)`

> **Note:** Only the types and modes of parameters are significant. The name of the parameter does not affect the signature.

If the signature dependency model is in effect, a dependency on a remote program unit causes an invalidation of the dependent unit if the dependent unit contains a call to a procedure in the parent unit, and the signature of this procedure has been changed in an incompatible manner. A program unit can be a package, stored procedure, stored function, or trigger.

Dependencies Among Other Remote Schema Objects

Oracle does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.

For example, assume that a local view is created and defined by a query that references a remote table. Also assume that a local procedure includes a SQL

statement that references the same remote table. Later, the definition of the table is altered.

As a result, the local view and procedure are never invalidated, even if the view or procedure is used after the table is altered, and even if the view or procedure now returns errors when used In this case, the view or procedure must be altered manually so errors are not returned. In such cases, lack of dependency management is preferable to unnecessary recompilations of dependent objects.

Dependencies of Applications

Code in database applications can reference objects in the connected database. For example, OCI, Precompiler, and SQL*Module applications can submit anonymous PL/SQL blocks. Triggers in Oracle Forms applications can reference a schema object.

Such applications are dependent on the schema objects they reference. Dependency management techniques vary, depending on the development environment. Refer to the appropriate manuals for your application development tools and your operating system for more information about managing the remote dependencies within database applications.

Part VI

Parallel SQL and Direct-Load INSERT

Part VI describes parallel execution of SQL statements and the direct-load INSERT feature. It contains the following chapters:

- Chapter 20, "Parallel Execution of SQL Statements"
- Chapter 21, "Direct-Path INSERT"

20

Parallel Execution of SQL Statements

This chapter describes the parallel execution of SQL statements. The topics in this chapter include:

- Introduction to Parallel Execution
- How Parallel Execution Works
- SQL Operations That Can Be Parallelized

> **Note:** The parallel execution features described in this chapter are available only if you have purchased the Oracle9*i* Enterprise Edition. See *Oracle9i Database New Features* for information about Oracle9*i* Enterprise Edition.

Introduction to Parallel Execution

When Oracle executes SQL statements in parallel, multiple processes work together simultaneously to execute a single SQL statement. By dividing the work necessary to execute a statement among multiple processes, Oracle can execute the statement more quickly than if only a single process executed it. This is called **parallel execution** or **parallel processing**.

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems (DSS) and data warehouses. Symmetric multiprocessing (SMP), clustered systems, and massively parallel systems (MPP) gain the largest performance benefits from parallel execution because statement processing can be split up among many CPUs on a single Oracle system. You can also implement parallel execution on certain types of online transaction processing (OLTP) and hybrid systems.

Parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time. An example of this is when 12 processes handle 12 different months in a year instead of one process handling all 12 months by itself. The improvement in performance can be quite high.

Parallel execution helps systems scale in performance by making optimal use of hardware resources. If your system's CPUs and disk controllers are already heavily loaded, you need to alleviate the system's load or increase these hardware resources before using parallel execution to improve performance.

Some tasks are not well-suited for parallel execution. For example, many OLTP operations are relatively fast, completing in mere seconds or fractions of seconds, and the overhead of utilizing parallel execution would be large, relative to the overall execution time.

> **See Also:** *Oracle9i Data Warehousing Guide* for specific information on tuning your parameter files and database to take full advantage of parallel execution

When to Implement Parallel Execution

During business hours, most OLTP systems should probably not use parallel execution. During off-hours, however, parallel execution can effectively process high-volume batch operations. For example, a bank can use parallelized batch programs to perform the millions of updates required to apply interest to accounts.

The most common example of using parallel execution is for DSS. Complex queries, such as those involving joins or searches of very large tables, are often best executed in parallel.

Parallel execution is useful for many types of operations that access significant amounts of data. Parallel execution improves performance for:

- Queries
- Creation of large indexes
- Bulk inserts, updates, and deletes
- Aggregations and copying

Parallel execution benefits systems that have *all* of the following characteristics:

- Symmetric multiprocessors (SMP), clusters, or massively parallel systems (for example, multiple CPUs)
- Sufficient I/O bandwidth
- Under-utilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes such as sorts, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution can reduce system performance on overutilized systems or systems with insufficient I/O bandwidth.

> **See Also:** *Oracle9i Data Warehousing Guide* for further information regarding when to implement parallel execution

When Not to Implement Parallel Execution

Parallel execution is not normally useful for:

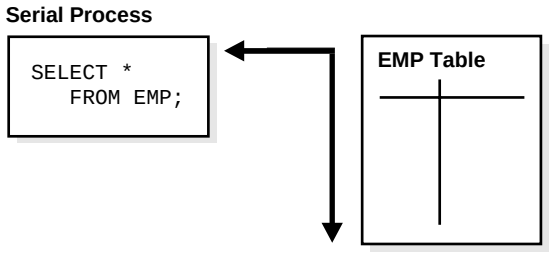
- Environments in which the typical query or transaction is very short (a few seconds or less). This includes most online transaction systems. Parallel execution is not useful in these environments because there is a cost associated with coordinating the parallel execution servers; for short transactions, the cost of this coordination may outweigh the benefits of parallelism.
- Environments in which the CPU, memory, and/or IO resources are already heavily utilized. Parallel execution is designed to exploit additional available

hardware resources; if no such resources are available, then parallel execution will not yield any benefits and indeed may be detrimental to performance.

How Parallel Execution Works

When parallel execution is being used, 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 emp`), one process performs the entire operation, as illustrated in Figure 20–1.

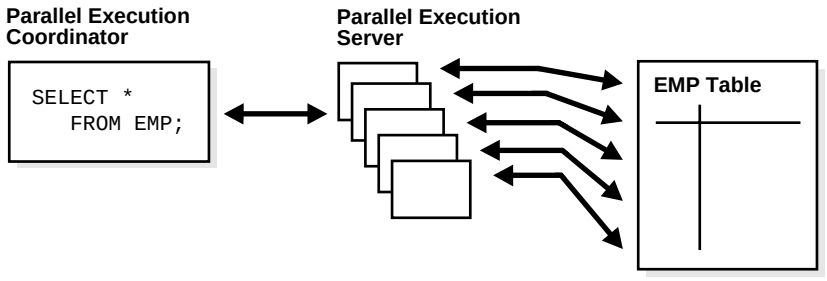
Figure 20–1 Serial Full Table Scan



Parallel execution performs these operations in parallel using multiple **parallel processes**. One process, known as the **parallel execution coordinator**, dispatches the execution of a statement to several **parallel execution servers** and coordinates the results from all of the server processes to send the results back to the user.

Figure 20–2 illustrates several parallel execution servers performing a scan of the table `emp`. The table is divided dynamically (**dynamic partitioning**) into load units called granules and each granule is read by a single parallel execution server. The granules are generated by the coordinator. Each granule is a range of physical blocks of the table emp. The mapping of granules to execution servers is not static, but is determined at execution time. When an execution server finishes reading the rows of the table `emp` corresponding to a granule, it gets another granule from the coordinator if there are any granules remaining. This continues till all granules are exhausted, in other words. the entire table `emp` has been read. The parallel execution servers send results back to the parallel execution coordinator, which assembles the pieces into the desired full table scan.

Figure 20–2 Parallel Full Table Scan



Given a query plan for a SQL query, the parallel execution coordinator breaks down each operator in a SQL query into parallel pieces, executes them in the right order as specified in the query, and then integrates the partial results produced by the parallel execution servers executing the operators. The number of parallel execution servers assigned to a single operation is the degree of parallelism (DOP) for an operation. Multiple operations within the same SQL statement all have the same degree of parallelism.

> **See Also:** *Oracle9i Data Warehousing Guide* for information regarding how Oracle divides work and handles DOP in multiuser environments

Parallelized SQL Statements

Each SQL statement undergoes an optimization and parallelization process when it is parsed. Therefore, when the data changes, if a more optimal execution plan or parallelization plan becomes available, Oracle can automatically adapt to the new situation.

After the optimizer determines the execution plan of a statement, the parallel execution coordinator determines the parallelization method for each operation in the execution plan. The coordinator must decide whether an operation can be performed in parallel and, if so, how many parallel execution servers to enlist. The number of parallel execution servers used for an operation is the degree of parallelism.

Parallelism Between Operations

To illustrate intra-operation parallelism and inter-operation parallelism, consider the following statement:

```
SELECT * FROM emp ORDER BY ename;
```

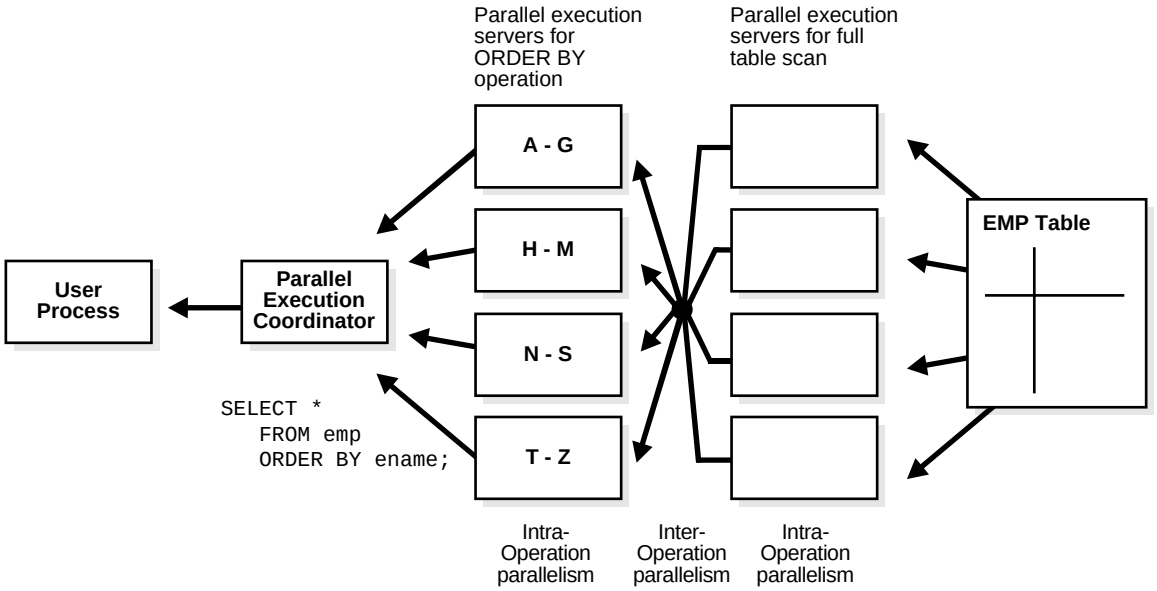
The execution plan implements a full scan of the `emp` table followed by a sorting of the retrieved rows based on the value of the `ename` column. For the sake of this example, assume the `ename` column is not indexed. Also assume that the degree of parallelism for the query is set to four, which means that four parallel execution servers can be active for any given operation.

Each of the two operations (scan and sort) performed concurrently is given its own set of parallel execution servers. Therefore, both operations have parallelism. Parallelization of an individual operation where the same operation is performed on smaller sets of rows by parallel execution servers achieves what is termed **intra-operation parallelism**. When two operations run concurrently on different sets of parallel execution servers with data flowing from one operation into the other, we achieve what is termed **inter-operation parallelism**.

Due to the producer/consumer nature of the Oracle server's operations, only two operations in a given tree need to be performed simultaneously to minimize execution time.

Figure 20–3 illustrates the parallel execution of our sample query.

Figure 20–3 Inter-Operation Parallelism and Dynamic Partitioning



As you can see from Figure 20–3, there are actually eight parallel execution servers involved in the query even though the degree of parallelism is four. This is because a parent and child operator can be performed at the same time (inter-operation parallelism).

Also note that all of the parallel execution servers involved in the scan operation send rows to the appropriate parallel execution server performing the sort operation. If a row scanned by a parallel execution server contains a value for the `ename` column between A and G, that row gets sent to the first `ORDER BY` parallel execution server. When the scan operation is complete, the sorting processes can return the sorted results to the coordinator, which in turn returns the complete query results to the user.

Note: When a set of parallel execution servers completes its operation, it moves on to operations higher in the data flow. For example, in the previous diagram, if there was another `ORDER BY` operation after the `ORDER BY`, the parallel execution servers performing the table scan perform the second `ORDER BY` operation after completing the table scan.

Degree of Parallelism

The parallel execution coordinator may enlist two or more of the instance's parallel execution servers to process a SQL statement. The number of parallel execution servers associated with a single operation is known as the **degree of parallelism**.

Note that the degree of parallelism applies directly only to intra-operation parallelism. If inter-operation parallelism is possible, the total number of parallel execution servers for a statement can be twice the specified degree of parallelism. No more than two sets of parallel execution servers can execute simultaneously. Each set of parallel execution servers may process multiple operations. Only two sets of parallel execution servers need to be active to guarantee optimal inter-operation parallelism.

Parallel execution is designed to effectively use multiple CPUs and disks to answer queries quickly. When multiple users use parallel execution at the same time, it is easy to quickly exhaust available CPU, memory, and disk resources.

Oracle provides several ways to manage resource utilization in conjunction with parallel execution environments, including:

- The adaptive multiuser algorithm, which reduces the degree of parallelism as the load on the system increases. You can turn this option with the `PARALLEL_ADAPTIVE_MULTI_USER` parameter of the `ALTER SYSTEM` statement or in your initialization parameter file.
- User resource limits and profiles, which allow you to set limits on the amount of various system resources available to each user as part of a user's security domain.
- The Database Resource Manager, which allows you to allocate resources to different groups of users.

See Also:

- *Oracle9i Database Reference* for information about PARALLEL_ADAPTIVE_MULTI_USER
- *Oracle9i SQL Reference* for the syntax of the ALTER SYSTEM SQL statement
- *Oracle9i Data Warehousing Guide*

Parallel Query Intra- and Inter-Operation Example

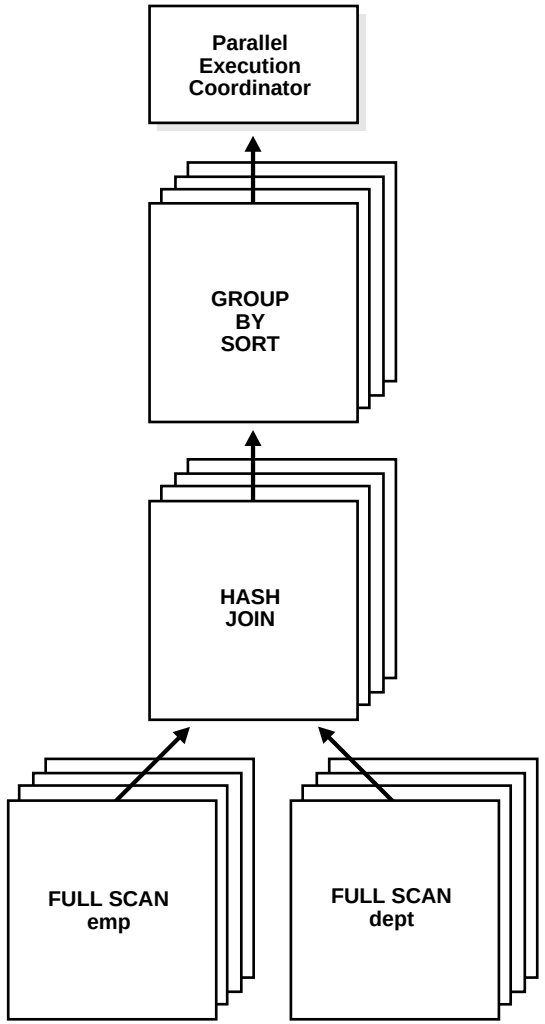
As an example of parallel query with intra- and inter-operation parallelism, consider a more complex query:

```
SELECT /*+ PARALLEL(emp 4) PARALLEL(dept 4) USE_HASH(emp) ORDERED */
       MAX(sal), AVG(sal)
FROM emp, dept
WHERE emp.deptno = dept.deptno
GROUP BY emp.deptno;
```

Note that hints have been used in the query to force the join order and join method, and to specify the degree of parallelism (DOP) of the tables emp and dept. In general, you should let the optimizer determine the order and method.

The query plan or data flow graph corresponding to this query is illustrated in Figure 20–4.

Figure 20–4 Data Flow Diagram for a Join of the EMP and DEPT Tables



Given two sets of parallel execution servers SS1 and SS2, the execution of this plan will proceed as follows: each server set (SS1 and SS2) will have four execution processes because of the PARALLEL hint in the query that specifies the DOP. In other words, the DOP will be four because each set of parallel execution servers will have four processes.

Slave set SS1 first scans the table emp while SS2 will fetch rows from SS1 and build a hash table on the rows. In other words, the parent servers in SS2 and the child servers in SS2 work concurrently: one in scanning emp in parallel, the other in consuming rows sent to it from SS1 and building the hash table for the hash join in parallel. This is an example of inter-operation parallelism.

After SS1 has finished scanning the entire table emp (that is, all granules or task units for emp are exhausted), it scans the table dept in parallel. It sends its rows to servers in SS2, which then perform the probes to finish the hash-join in parallel. After SS1 is done scanning the table dept in parallel and sending the rows to SS2, it switches to performing the GROUP BY in parallel. This is how two server sets run concurrently to achieve inter-operation parallelism across various operators in the query tree while achieving intra-operation parallelism in executing each operation in parallel.

Another important aspect of parallel execution is the re-partitioning of rows while they are sent from servers in one server set to another. For the query plan in Figure 20–4, after a server process in SS1 scans a row of emp, which server process of SS2 should it send it to? The partitioning of rows flowing up the query tree is decided by the operator into which the rows are flowing into. In this case, the partitioning of rows flowing up from SS1 performing the parallel scan of emp into SS2 performing the parallel hash-join is done by hash partitioning on the join column value. That is, a server process scanning emp computes a hash function of the value of the column emp.empno to decide the number of the server process in SS2 to send it to. The partitioning method used in parallel queries is explicitly shown in the EXPLAIN PLAN of the query. Note that the partitioning of rows being sent between sets of execution servers should not be confused with Oracle's partitioning feature whereby tables can be partitioned using hash, range, and other methods.

> **See Also:** *Oracle9i Data Warehousing Guide* for examples of using EXPLAIN PLAN with parallel query

SQL Operations That Can Be Parallelized

Most operations can be parallelized. The following are commonly parallelized to improve performance:

- Parallel Query
- Parallel DDL
- Parallel DML
- SQL*Loader

> **See Also:** *Oracle9i Data Warehousing Guide* for specific information regarding restrictions for parallel DML as well as some considerations to keep in mind when designing a warehouse

Parallel Query

You can parallelize queries and subqueries in `SELECT` statements, as well as the query portions of DDL statements and DML statements (`INSERT`, `UPDATE`, and `DELETE`). However, you cannot parallelize the query portion of a DDL or DML statement if it references a remote object. When you issue a parallel DML or DDL statement in which the query portion references a remote object, the operation is automatically executed serially.

> **See Also:** *Oracle9i SQL Reference* for information about the syntax and restrictions for parallel query statements

Parallel DDL

You can normally use parallel DDL where you use regular DDL. There are, however, some additional details to consider when designing your database. One important restriction is that parallel DDL cannot be used on tables with object or `LOB` columns.

DDL Statements that can be Parallelized

You can parallelize the `CREATE TABLE AS SELECT`, `CREATE INDEX`, and `ALTER INDEX REBUILD` statements. If the table is partitioned, you can parallelize `ALTER TABLE MOVE` or `[SPLIT` or `COALESCE]` statements as well. You can also use parallelism for `ALTER INDEX REBUILD [`or `SPLIT]` when the index is partitioned.

All of these DDL operations can be performed in no-logging mode for either parallel or serial execution.

The `CREATE TABLE` statement for an index-organized table can be parallelized either with or without an `AS SELECT` clause.

Different parallelism is used for different operations. Parallel create (partitioned) table as select and parallel create (partitioned) index execute with a degree of parallelism equal to the number of partitions.

Parallel operations require accurate statistics to perform optimally.

See Also:

- *Oracle9i SQL Reference* for information about the syntax and restrictions for parallel DDL statements
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about `LOB` restrictions

Parallel DML

Parallel DML (parallel insert, update, and delete) uses parallel execution mechanisms to speed up or scale up large DML operations against large database tables and indexes. You can also use `INSERT ... SELECT` statements to insert rows into multiple tables as part of a single DML statement. You can normally use parallel DML where you use regular DML.

Although data manipulation language (DML) normally includes queries, the term parallel DML refers only to inserts, updates, upserts and deletes done in parallel.

See Also:

- *Oracle9i SQL Reference* for information about the syntax and restrictions for parallel DML statements
- *Oracle9i Data Warehousing Guide* for specific information regarding restrictions for parallel DML as well as some considerations to keep in mind when designing a warehouse

SQL*Loader

You can parallelize the use of SQL*Loader, where large amounts of data are routinely encountered. To speed up your loads, you can use a parallel direct-path load as in the following example:

```
SQLLOAD USERID=SCOTT/TIGER CONTROL=LOAD1.CTL DIRECT=TRUE PARALLEL=TRUE
SQLLOAD USERID=SCOTT/TIGER CONTROL=LOAD2.CTL DIRECT=TRUE PARALLEL=TRUE
SQLLOAD USERID=SCOTT/TIGER CONTROL=LOAD3.CTL DIRECT=TRUE PARALLEL=TRUE
```

You can also use a parameter file to achieve the same thing.

An important point to remember is that indexes are not maintained during a parallel load.

> **See Also:** *Oracle9i Database Utilities* for information about the syntax and restrictions for parallel loading

How to Make a Statement Run in Parallel

The way you make a statement run in parallel depends upon the type of parallel operation. The three types of parallel operation are:

- Parallel Query
- Parallel DDL
- Parallel DML

Parallel Query

To achieve parallelism for SQL query statements, one or more of the tables being scanned should have a parallel attribute.

Parallel DDL

To achieve parallelism for SQL DDL statements, the parallel clause should be specified.

Parallel DML

Due to the differences in locking between serial and parallel DML, you must explicitly enable parallel DML before you can use it. To achieve parallelism for SQL DML statements, you must first enable parallel DML in your session:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Then any DML issued against a table with a parallel attribute will occur in parallel, if no PDML restrictions are violated. For example:

```
INSERT INTO mytable SELECT * FROM origtable;
```

See Also:

- *Oracle9i SQL Reference* for information about the syntax to implement parallelism
- *Oracle9i Database Reference* for information about the syntax to implement parallelism from a parameter file
- *Oracle9i Data Warehousing Guide* for specific information regarding restrictions for parallel DML

21

Direct-Path INSERT

This chapter describes the Oracle direct-path `INSERT` feature for serial or parallel inserts. It also describes the `NOLOGGING` feature that is available for direct-path `INSERT` and some DDL statements. This chapter's topics include:

- Introduction to Direct-Path INSERT
- Advantages of Direct-Path INSERT
- Serial and Parallel Direct-Path INSERT
- Direct-Path INSERT Into Partitioned and Nonpartitioned Tables
- Direct-Path INSERT and Logging Mode
- Additional Considerations for Direct-Path INSERT

Note: The parallel direct-path `INSERT` feature described in this chapter is available only if you have purchased the Oracle9*i* Enterprise Edition. See *Oracle9i Database New Features* for more information.

See Also:

- Chapter 20, "Parallel Execution of SQL Statements" for more information about parallel execution `INSERT` issues
- *Oracle9i Data Warehousing Guide*

Introduction to Direct-Path INSERT

Oracle inserts data into a table in one of two ways:

- During **conventional insert operations**, Oracle reuses free space in the table, interleaving newly inserted data with existing data. During such operations, Oracle also maintains referential integrity constraints.
- During **direct-path insert operations**, Oracle appends the inserted data after existing data in the table. Data is written directly into datafiles, bypassing the buffer cache. Free space in the existing data is not reused, and referential integrity constraints are ignored. These procedures combined can enhance performance.

You can implement direct-path insert operations by using direct-path `INSERT` statements or by using Oracle's direct-path loader utility, SQL*Loader. This section discusses direct-path `INSERT`. For information on direct-path load, please refer to the documentation on SQL*Loader in *Oracle9i Database Utilities.*

Note: Direct-path `INSERT` is subject to a number of restrictions. For a listing of these restriction, please refer to *Oracle9i SQL Reference.*

Advantages of Direct-Path INSERT

The performance benefits of direct-path `INSERT` arise from several factors:

- During direct-path `INSERT`, you can disable the logging of redo and undo entries. Conventional insert operations, in contrast, must always log such entries, because those operations reuse free space and maintain referential integrity.
- To create a new table with data from an existing table, you have the choice of creating the new table and then inserting into it, or executing a `CREATE TABLE ... AS SELECT statement`. By creating the table and then using direct-path `INSERT` operations, you update any indexes defined on the target table during the insert operation. The table resulting from a `CREATE TABLE ... AS SELECT` statement, in contrast, does not have any indexes defined on it; you must define them later.
- Direct-path `INSERT` operations ensure atomicity of the transaction, even when run in parallel mode. Atomicity cannot be guaranteed during parallel direct-path loads (using SQL*Loader).

- If errors occur during parallel direct-path loads, some indexes may be marked UNUSABLE at the end of the load. Parallel direct-path INSERT, in contrast, rolls back the statement if errors occur during index update.

Serial and Parallel Direct-Path INSERT

When you are inserting in parallel DML mode, direct-path INSERT is the default. In order to be running in parallel DML mode, the following requirements must be met:

- You must have Oracle Enterprise Edition installed.
- You must enable parallel DML in your session. To do this, execute the following statement:

  ```
  ALTER SESSION { ENABLE | FORCE } PARALLEL DML;
  ```

- You must have specified the parallel attribute for the target table, either at create time or subsequently, or you must specify the PARALLEL hint for each insert operation.

You can disable direct-path INSERT by specifying the NOAPPEND hint in each INSERT statement. Doing so overrides parallel DML mode.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for more information on using hints

When you are inserting in serial mode, you must activate direct-path INSERT by specifying the APPEND hint in each insert statement, either immediately after the INSERT keyword, or immediately after the SELECT keyword in the subquery of the INSERT statement.

Note: Direct-path INSERT supports only the subquery syntax of the INSERT statement, not the VALUES clause. For more information on the subquery syntax of INSERT statements, refer to *Oracle9i SQL Reference.*

Direct-Path INSERT Into Partitioned and Nonpartitioned Tables

You can use direct-path `INSERT` on both partitioned and nonpartitioned tables.

Serial Direct-Path Insert into Partitioned and Nonpartitioned Tables

The single process inserts data beyond the current high water mark of the table segment or of each partition segment. (The **high-water mark** is the level at which blocks have never been formatted to receive data.) When a `COMMIT` executes, the high-water mark is updated to the new value, making the data visible to users.

Parallel Direct-Path INSERT into Partitioned Tables

This situation is analogous to serial direct-path `INSERT`. Each parallel execution server is assigned one or more partitions, with no more than one process working on a single partition. Each parallel execution server inserts data beyond the current high-water mark of its assigned partition segment(s). When a `COMMIT` executes, the high-water mark of each partition segment is updated to its new value, making the data visible to users.

Parallel Direct-Path INSERT into a Nonpartitioned Table

Each parallel execution server allocates a new temporary segment and inserts data into that temporary segment. When a `COMMIT` executes, the parallel execution coordinator merges the new temporary segments into the primary table segment, where it is visible to users.

Direct-Path INSERT and Logging Mode

Direct-path `INSERT` lets you choose whether to log redo and undo information during the insert operation.

- You can specify logging mode for a table, partition, index, or `LOB` storage at create time (in a `CREATE` statement) or subsequently (in an `ALTER` statement).
- If you do not specify either `LOGGING` or `NOLOGGING` at these times:
  - The logging attribute of a partition defaults to the logging attribute of its table.
  - The logging attribute of a table or index defaults to the logging attribute of the tablespace in which it resides.

  - The logging attribute of LOB storage defaults to LOGGING if you specify CACHE for LOB storage. If you do not specify CACHE, the logging attributes defaults to that of the tablespace in which the LOB values resides.
- You set the logging attribute of a tablespace in a CREATE TABLESPACE or ALTER TABLESPACE statements.

Direct-Path INSERT with Logging

In this mode, Oracle performs full redo logging for instance and media recovery. If the database is in ARCHIVELOG mode, you can archive online redo logs to tape. If the database is in NOARCHIVELOG mode, you can recover instance crashes but not disk failures.

Direct-Path INSERT without Logging

In this mode, Oracle inserts data without redo or undo logging. (Some minimal logging is done to mark new extents invalid, and data dictionary changes are always logged.) This mode improves performance. However, if you subsequently must perform media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because no redo data was logged for them. Therefore, it is important that you back up the data after such an insert operation.

See Also:

- *Oracle9i Backup and Recovery Concepts* for recovery information
- *Oracle9i SQL Reference* for information on logging mode in operations other than inserts

Additional Considerations for Direct-Path INSERT

Index Maintenance

Oracle performs index maintenance at the end of direct-path INSERT operations on tables (partitioned or nonpartitioned) that have indexes. This index maintenance is performed by the parallel execution servers for parallel direct-path INSERT or by the single process for serial direct-path INSERT. You can avoid the performance impact of index maintenance by dropping the index before the INSERT operation and then rebuilding it afterward.

Space Considerations

Direct-path `INSERT` requires more space than conventional-path `INSERT`, because direct-path `INSERT` does not use existing space in the free lists of the segment.

All serial direct-path `INSERT` operations as well as parallel direct-path `INSERT` into partitioned tables insert data above the high-water mark of the affected segment. This requires some additional space.

Parallel direct-path `INSERT` into nonpartitioned tables requires even more space, because it creates a temporary segment for each degree of parallelism. If the nonpartitioned table is not in a locally managed tablespace in automatic mode, you can modify the values of the `NEXT` and `PCTINCREASE` storage parameter and `MINIMUM EXTENT` tablespace parameter to provide sufficient (but not excess) storage for the temporary segments. Choose values for these parameters so that:

- The size of each extent is not too small (no less than 1 MB). This setting affects the total number of extents in the object.
- The size of each extent is not so large that the parallel `INSERT` results in wasted space on segments that are larger than necessary.

> **See Also:** *Oracle9i SQL Reference* for information on setting these parameters

After the direct-path `INSERT` operation is complete, you can reset these parameters to settings more appropriate for serial operations.

Locking Considerations

During direct-path `INSERT`, Oracle obtains exclusive locks on the table (or on all partitions of a partitioned table). As a result, users cannot perform any concurrent insert, update, or delete operations on the table, and concurrent index creation and build operations are not permitted. Concurrent queries, however, are supported, but the query will return only the information before the insert operation.

Part VII

Data Protection

Part VII describes how Oracle protects the data in a database and explains what the database administrator can do to provide additional protection for data.

Part VII contains the following chapters:

- Chapter 22, "Data Concurrency and Consistency"
- Chapter 23, "Data Integrity"
- Chapter 24, "Controlling Database Access"
- Chapter 25, "Privileges, Roles, and Security Policies"
- Chapter 26, "Auditing"

22

Data Concurrency and Consistency

This chapter explains how Oracle maintains consistent data in a multiuser database environment. The chapter includes:

- Introduction to Data Concurrency and Consistency in a Multiuser Environment
- How Oracle Manages Data Concurrency and Consistency
- How Oracle Locks Data
- Flashback Query

Introduction to Data Concurrency and Consistency in a Multiuser Environment

In a single-user database, the user can modify data in the database without concern for other users modifying the same data at the same time. However, in a multiuser database, the statements within multiple simultaneous transactions can update the same data. Transactions executing at the same time need to produce meaningful and consistent results. Therefore, control of data concurrency and data consistency is vital in a multiuser database.

- **Data concurrency** means that many users can access data at the same time.
- **Data consistency** means that each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users.

To describe consistent transaction behavior when transactions execute at the same time, database researchers have defined a transaction isolation model called **serializability**. The serializable mode of transaction behavior tries to ensure that transactions execute in such a way that they appear to be executed one at a time, or serially, rather than concurrently.

While this degree of isolation between transactions is generally desirable, running many applications in this mode can seriously compromise application throughput. Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insert into a table being queried by another transaction. In short, real-world considerations usually require a compromise between perfect transaction isolation and performance.

Oracle offers two isolation levels, providing application developers with operational modes that preserve consistency and provide high performance.

> **See Also:** Chapter 23, "Data Integrity" for information about data integrity, which enforces business rules associated with a database

Preventable Phenomena and Transaction Isolation Levels

The ANSI/ISO SQL standard (SQL92) defines four levels of transaction isolation with differing degrees of impact on transaction processing throughput. These isolation levels are defined in terms of three phenomena that must be prevented between concurrently executing transactions.

The three preventable phenomena are:

dirty read	A transaction reads data that has been written by another transaction that has not been committed yet.
nonrepeatable (fuzzy) read	A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.
phantom read	A transaction re-executes 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.

SQL92 defines four levels of isolation in terms of the phenomena a transaction running at a particular isolation level is permitted to experience. They are shown in Table 22–1:

Table 22–1 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 offers the read committed and serializable isolation levels, as well as a read-only mode that is not part of SQL92. Read committed is the default.

> **See Also:** "How Oracle Manages Data Concurrency and Consistency" on page 22-4 for a full discussion of read committed and serializable isolation levels

Overview of 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.

Resources include two general types of objects:

- User objects, such as tables and rows (structures and data)
- System objects not visible to users, such as shared data structures in the memory and data dictionary rows

> **See Also:** "How Oracle Locks Data" on page 22-16 for more information about locks

How Oracle Manages Data Concurrency and Consistency

Oracle maintains data consistency in a multiuser environment by using a multiversion consistency model and various types of locks and transactions. The following topics are discussed in this section:

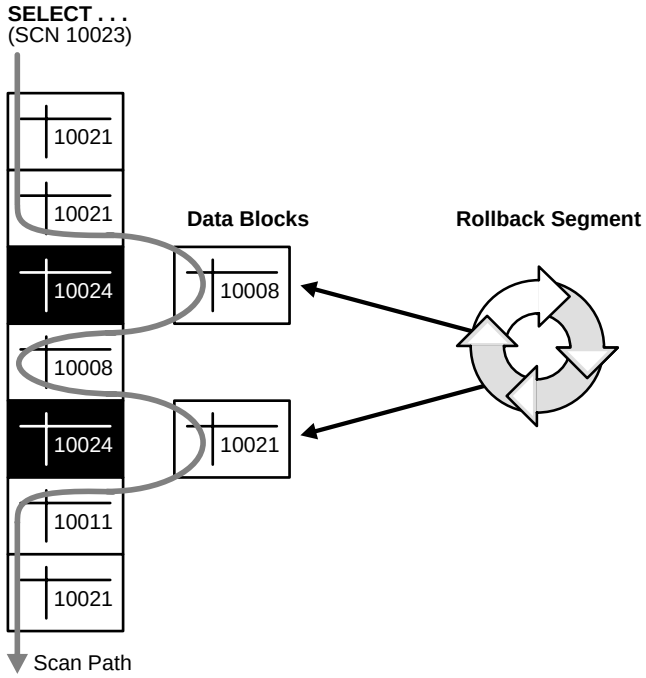
- Multiversion Concurrency Control
- Statement-Level Read Consistency
- Transaction-Level Read Consistency
- Read Consistency with Oracle9i Real Application Clusters
- Oracle Isolation Levels
- Comparison of Read Committed and Serializable Isolation
- Choice of Isolation Level

Multiversion Concurrency Control

Oracle automatically provides read consistency to a query so that all the data that the query sees comes from a single point in time *(***statement-level read consistency***)*. Oracle can also provide read consistency to all of the queries in a transaction *(***transaction-level read consistency***)*.

Oracle uses the information maintained in its rollback segments to provide these consistent views. The rollback segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Figure 22–1 shows how Oracle provides statement-level read consistency using data in rollback segments.

Figure 22–1 Transactions and Read Consistency



As a query enters the execution stage, the current system change number (SCN) is determined. In Figure 22–1, this system change number is 10023. As data blocks are read on behalf of the query, only blocks written with the observed SCN are used. Blocks with changed data (more recent SCNs) are reconstructed from data in the rollback segments, and the reconstructed data is returned for the query. Therefore, each query returns all committed data with respect to the SCN recorded at the time that query execution began. Changes of other transactions that occur during a query's execution are not observed, guaranteeing that consistent data is returned for each query.

Statement-Level Read Consistency

Oracle always enforces **statement-level** read consistency. This guarantees that all the data returned by a single query comes from a single point in time—the time that the query began. Therefore, a query never sees dirty data nor any of the changes made by transactions that commit during query execution. As query execution

proceeds, only data committed before the query began is visible to the query. The query does not see changes committed after statement execution begins.

A consistent result set is provided for every query, guaranteeing data consistency, with no action on the user's part. The SQL statements `SELECT`, `INSERT` with a subquery, `UPDATE`, and `DELETE` all query data, either explicitly or implicitly, and all return consistent data. Each of these statements uses a query to determine which data it will affect (`SELECT`, `INSERT`, `UPDATE`, or `DELETE`, respectively).

A `SELECT` statement is an explicit query and can have nested queries or a join operation. An `INSERT` statement can use nested queries. `UPDATE` and `DELETE` statements can use `WHERE` clauses or subqueries to affect only some rows in a table rather than all rows.

Queries used in `INSERT`, `UPDATE`, and `DELETE` statements are guaranteed a consistent set of results. However, they do not see the changes made by the DML statement itself. In other words, the query in these operations sees data as it existed before the operation began to make changes.

Transaction-Level Read Consistency

Oracle also offers the option of enforcing **transaction-level read consistency**. When a transaction executes in serializable mode, all data accesses reflect the state of the database as of the time the transaction began. This means that the data seen by all queries within the same transaction is consistent with respect to a single point in time, except that queries made by a serializable transaction do see changes made by the transaction itself. Transaction-level read consistency produces repeatable reads and does not expose a query to phantoms.

Read Consistency with Oracle9*i* Real Application Clusters

Oracle9*i* Real Application Clusters uses a cache-to-cache block transfer mechanism known as Cache Fusion to transfer read-consistent images of blocks from one instance to another. Real Application Clusters does this using high speed, low latency interconnects to satisfy remote requests for data blocks.

> **See Also:** *Oracle9i Real Application Clusters Concepts* for more information

Oracle Isolation Levels

Oracle provides these transaction isolation levels:

Read committed This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle query never reads dirty (uncommitted) data.

Because Oracle does not prevent other transactions from modifying the data read by a query, that data can be changed by other transactions between two executions of the query. Thus, a transaction that executes a given query twice can experience both nonrepeatable read and phantoms.

Serializable Serializable transactions see only those changes that were committed at the time the transaction began, plus those changes made by the transaction itself through `INSERT`, `UPDATE`, and `DELETE` statements. Serializable transactions do not experience nonrepeatable reads or phantoms.

Read-only Read-only transactions see only those changes that were committed at the time the transaction began and do not allow `INSERT`, `UPDATE`, and `DELETE` statements.

Set the Isolation Level

Application designers, application developers, and database administrators can choose appropriate isolation levels for different transactions, depending on the application and workload. You can set the isolation level of a transaction by using one of these statements at the beginning of a transaction:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SET TRANSACTION ISOLATION LEVEL READ ONLY;
```

To save the networking and processing cost of beginning each transaction with a `SET TRANSACTION` statement, you can use the `ALTER SESSION` statement to set the transaction isolation level for all subsequent transactions:

```
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;

ALTER SESSION SET ISOLATION_LEVEL READ COMMITTED;
```

See Also: *Oracle9i SQL Reference* for detailed information on any of these SQL statements

Read Committed Isolation

The default isolation level for Oracle is read committed. This degree of isolation is appropriate for environments where few transactions are likely to conflict. Oracle causes each query to execute with respect to its own materialized view time, thereby permitting nonrepeatable reads and phantoms for multiple executions of a query, but providing higher potential throughput. Read committed isolation is the appropriate level of isolation for environments where few transactions are likely to conflict.

Serializable Isolation

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

Serializable isolation permits concurrent transactions to make only those database changes they could have made if the transactions had been scheduled to execute one after another. Specifically, Oracle permits a serializable transaction to modify a data row only if it can determine that prior changes to the row were made by transactions that had committed when the serializable transaction began.

To make this determination efficiently, Oracle uses control information stored in the data block that indicates which rows in the block contain committed and uncommitted changes. In a sense, the block contains a recent history of transactions that affected each row in the block. The amount of history that is retained is controlled by the `INITRANS` parameter of `CREATE TABLE` and `ALTER TABLE`.

Under some circumstances, Oracle can have insufficient history information to determine whether a row has been updated by a "too recent" transaction. This can occur when many transactions concurrently modify the same data block, or do so in a very short period. You can avoid this situation by setting higher values of `INITRANS` for tables that will experience many transactions updating the same blocks. Doing so enables Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block.

Oracle generates an error when a serializable transaction tries to update or delete data modified by a transaction that commits *after* the serializable transaction began:

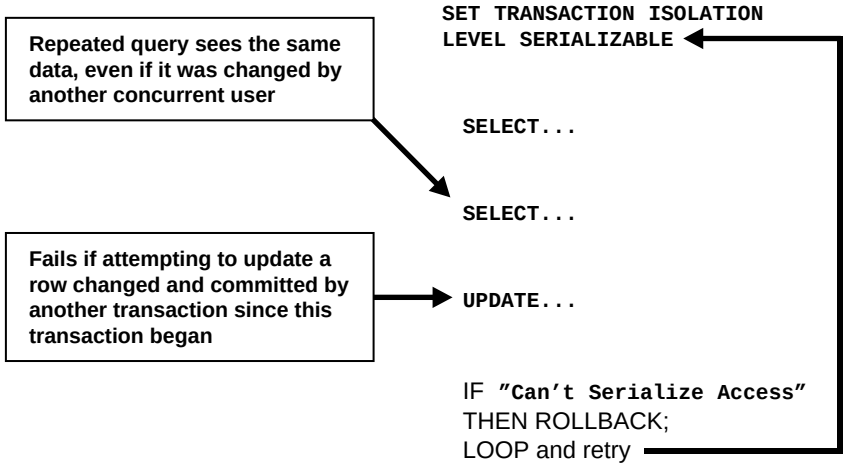
```
ORA-08177: Cannot serialize access for this transaction
```

When a serializable transaction fails with the "Cannot serialize access" error, the application can take any of several actions:

- 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

Figure 22–2 shows an example of an application that rolls back and retries the transaction after it fails with the "Cannot serialize access" error:

Figure 22–2 Serializable Transaction Failure



Comparison of Read Committed and Serializable Isolation

Oracle gives the application developer a choice of two transaction isolation levels with different characteristics. Both the read committed and serializable isolation levels provide a high degree of consistency and concurrency. Both levels provide the contention-reducing benefits of Oracle's read consistency multiversion concurrency

control model and exclusive row-level locking implementation and are designed for real-world application deployment.

Transaction Set Consistency

A useful way to view the read committed and serializable isolation levels in Oracle is to consider the following scenario: Assume you have a collection of database tables (or any set of data), a particular sequence of reads of rows in those tables, and the set of transactions committed at any particular time. An operation (a query or a transaction) is **transaction set consistent** if all its reads return data written by the same set of committed transactions. An operation is not transaction set consistent if some reads reflect the changes of one set of transactions and other reads reflect changes made by other transactions. An operation that is not transaction set consistent in effect sees the database in a state that reflects no single set of committed transactions.

Oracle provides transactions executing in read committed mode with transaction set consistency on a per-statement basis. Serializable mode provides transaction set consistency on a per-transaction basis.

Table 22–2 summarizes key differences between read committed and serializable transactions in Oracle.

Table 22–2 Read Committed and Serializable Transactions

	Read Committed	Serializable
Dirty write	Not possible	Not possible
Dirty read	Not possible	Not possible
Nonrepeatable read	Possible	Not possible
Phantoms	Possible	Not possible
Compliant with ANSI/ISO SQL 92	Yes	Yes
Read materialized view time	Statement	Transaction
Transaction set consistency	Statement level	Transaction level
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes

Table 22–2 Read Committed and Serializable Transactions

	Read Committed	Serializable
Waits for blocking transaction	Yes	Yes
Subject to "cannot serialize access"	No	Yes
Error after blocking transaction aborts	No	No
Error after blocking transaction commits	No	Yes

Row-Level Locking

Both read committed and serializable transactions use row-level locking, and both will wait if they try to change a row updated by an uncommitted concurrent transaction. The second transaction that tries to update a given row waits for the other transaction to commit or roll back and release its lock. If that other transaction rolls back, the waiting transaction, regardless of its isolation mode, can proceed to change the previously locked row as if the other transaction had not existed.

However, if the other blocking transaction commits and releases its locks, a read committed transaction proceeds with its intended update. A serializable transaction, however, fails with the error "Cannot serialize access", because the other transaction has committed a change that was made since the serializable transaction began.

Referential Integrity

Because Oracle does not use read locks in either read-consistent or serializable transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level cannot assume that the data they read will remain unchanged during the execution of the transaction even though such changes are not visible to the transaction. Database inconsistencies can result unless such application-level consistency checks are coded with this in mind, even when using serializable transactions.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about referential integrity and serializable transactions

---

Note: You can use both read committed and serializable transaction isolation levels with Oracle9*i* Real Application Clusters.

Distributed Transactions

In a distributed database environment, a given transaction updates data in multiple physical databases protected by two-phase commit to ensure all nodes or none commit. In such an environment, all servers, whether Oracle or non-Oracle, that participate in a **serializable** transaction are required to support serializable isolation mode.

If a serializable transaction tries to update data in a database managed by a server that does not support serializable transactions, the transaction receives an error. The transaction can roll back and retry only when the remote server does support serializable transactions.

In contrast, **read committed** transactions can perform distributed transactions with servers that do not support serializable transactions.

See Also: *Oracle9i Database Administrator's Guide*

Choice of Isolation Level

Application designers and developers should choose an isolation level based on application performance and consistency needs as well as application coding requirements.

For environments with many concurrent users rapidly submitting transactions, designers must assess transaction performance requirements in terms of the expected transaction arrival rate and response time demands. Frequently, for high-performance environments, the choice of isolation levels involves a trade-off between consistency and concurrency.

Application logic that checks database consistency must take into account the fact that reads do not block writes in either mode.

Oracle isolation modes provide high levels of consistency, concurrency, and performance through the combination of row-level locking and Oracle's multiversion concurrency control system. Readers and writers do not block one another in Oracle. Therefore, while queries still see consistent data, both read committed and serializable isolation provide a high level of concurrency for high performance, without the need for reading uncommitted ("dirty") data.

Read Committed Isolation

For many applications, read committed is the most appropriate isolation level. Read committed isolation can provide considerably more concurrency with a somewhat

increased risk of inconsistent results due to phantoms and non-repeatable reads for some transactions.

Many high-performance environments with high transaction arrival rates require more throughput and faster response times than can be achieved with serializable isolation. Other environments that supports users with a very low transaction arrival rate also face very low risk of incorrect results due to phantoms and nonrepeatable reads. Read committed isolation is suitable for both of these environments.

Oracle read committed isolation provides transaction set consistency for every query. That is, every query sees data in a consistent state. Therefore, read committed isolation will suffice for many applications that might require a higher degree of isolation if run on other database management systems that do not use multiversion concurrency control.

Read committed isolation mode does not require application logic to trap the "Cannot serialize access" error and loop back to restart a transaction. In most applications, few transactions have a functional need to issue the same query twice, so for many applications protection against phantoms and non-repeatable reads is not important. Therefore many developers choose read committed to avoid the need to write such error checking and retry code in each transaction.

Serializable Isolation

Oracle's serializable isolation is suitable for environments where there is a relatively low chance that two concurrent transactions will modify the same rows and the long-running transactions are primarily read-only. It is most suitable for environments with large databases and short transactions that update only a few rows.

Serializable isolation mode provides somewhat more consistency by protecting against phantoms and nonrepeatable reads and can be important where a read/write transaction executes a query more than once.

Unlike other implementations of serializable isolation, which lock blocks for read as well as write, Oracle provides nonblocking queries and the fine granularity of row-level locking, both of which reduce write/write contention. For applications that experience mostly read/write contention, Oracle serializable isolation can provide significantly more throughput than other systems. Therefore, some applications might be suitable for serializable isolation on Oracle but not on other systems.

All queries in an Oracle serializable transaction see the database as of a single point in time, so this isolation level is suitable where multiple consistent queries must be issued in a read/write transaction. A report-writing application that generates summary data and stores it in the database might use serializable mode because it provides the consistency that a `READ ONLY` transaction provides, but also allows `INSERT`, `UPDATE`, and `DELETE`.

> **Note:** Transactions containing DML statements with subqueries should use serializable isolation to guarantee consistent read.

Coding serializable transactions requires extra work by the application developer to check for the "Cannot serialize access" error and to roll back and retry the transaction. Similar extra coding is needed in other database management systems to manage deadlocks. For adherence to corporate standards or for applications that are run on multiple database management systems, it may be necessary to design transactions for serializable mode. Transactions that check for serializability failures and retry can be used with Oracle read committed mode, which does not generate serializability errors.

Serializable mode is probably not the best choice in an environment with relatively long transactions that must update the same rows accessed by a high volume of short update transactions. Because a longer running transaction is unlikely to be the first to modify a given row, it will repeatedly need to roll back, wasting work. Note that a conventional read-locking, pessimistic implementation of serializable mode would not be suitable for this environment either, because long-running transactions—even read transactions—would block the progress of short update transactions and vice versa.)

Application developers should take into account the cost of rolling back and retrying transactions when using serializable mode. As with read-locking systems, where deadlocks occur frequently, use of serializable mode requires rolling back the work done by aborted transactions and retrying them. In a high contention environment, this activity can use significant resources.

In most environments, a transaction that restarts after receiving the "Cannot serialize access" error is unlikely to encounter a second conflict with another transaction. For this reason it can help to execute those statements most likely to contend with other transactions as early as possible in a serializable transaction. However, there is no guarantee that the transaction will complete successfully, so the application should be coded to limit the number of retries.

Although Oracle serializable mode is compatible with SQL92 and offers many benefits compared with read-locking implementations, it does not provide semantics identical to such systems. Application designers must take into account the fact that reads in Oracle do not block writes as they do in other systems. Transactions that check for database consistency at the application level can require coding techniques such as the use of `SELECT FOR UPDATE`. This issue should be considered when applications using serializable mode are ported to Oracle from other environments.

Quiesce Database

Oracle9*i*, Release 1 (9.0.1), enables a database administrator to put the system into **quiesced state**. The system is in quiesced state if there are no active sessions, other than SYS and SYSTEM. An active session is defined as a session that is currently inside a transaction, a query, a fetch or a PL/SQL procedure, or a session that is currently holding any shared resources (e.g. enqueues). Database administrators are the only users who can proceed when the system is in quiesced state.

DBAs can perform certain actions in the quiesced state that cannot be safely done when the system is not quiesced. These actions include:

- Actions that might fail if there are concurrent user transactions or queries. For example, changing the schema of a database table will fail if a concurrent transaction is accessing the same table.
- Actions whose intermediate effect could be detrimental to concurrent user transactions or queries. For example:

  1. Change the schema of a database table.
  2. Update a PL/SQL procedure to a new version that uses this new schema of the database table.

  Between Step 1 and Step 2, the new schema of the table is inconsistent with the implementation of the PL/SQL procedure. This inconsistency would adversely affect users concurrently trying to execute the PL/SQL procedure.

For systems that must operate continuously, the ability to perform such actions without shutting down the database is critical.

The Database Resource Manager blocks all actions that were initiated by a user other than SYS or SYSTEM while the system is quiesced. Such actions are allowed to proceed when the system goes back to normal (unquiesced) state. Users do not get any additional error messages from the quiesced state.

How a Database Is Quiesced The DBA uses the `ALTER SYSTEM QUIESCE RESTRICTED` command to quiesce the database. Only users SYS and SYSTEM can issue the `ALTER SYSTEM QUIESCE RESTRICTED` command. For all instances with the database open, issuing this statement has the following effect:

- Oracle instructs the Database Resource Manager in all instances to prevent all inactive sessions (other than SYS and SYSTEM) from becoming active. No user other than SYS and SYSTEM can start a new transaction, a new query, a new fetch, or a new PL/SQL operation.
- Oracle waits for all existing transactions in all instances that were initiated by a user other than SYS or SYSTEM to finish (either commit or abort). Oracle also waits for all running queries, fetches, and PL/SQL procedures in all instances that were initiated by users other than SYS or SYSTEM and that are not inside transactions to finish. If a query is carried out by multiple successive OCI fetches, Oracle does not wait for all fetches to finish. It waits for the current fetch to finish and then blocks the next fetch. Oracle also waits for all sessions (other than those of SYS or SYSTEM) that hold any shared resources (such as enqueues) to release those resources. After all these operations finish, Oracle places the database into quiesced state and finishes executing the QUIESCE RESTRICTED statement.
- If an instance is running in shared server mode, Oracle instructs the Database Resource Manager to block logins (other than SYS or SYSTEM) on that instance. If an instance is running in non-shared-server mode, Oracle does not impose any restrictions on user logins in that instance.

During the quiesced state, you cannot change the Resource Manager plan in any instance.

The `ALTER SYSTEM UNQUIESCE` command puts all running instances back into normal mode, so that all blocked actions can proceed.

See Also:

- *Oracle9i SQL Reference*
- *Oracle9i Database Administrator's Guide*

How Oracle Locks Data

Locks are mechanisms that prevent destructive interaction between transactions accessing the same **resource**—either user objects such as tables and rows or system objects not visible to users, such as shared data structures in memory and data dictionary rows.

In all cases, Oracle automatically obtains necessary locks when executing SQL statements, so users need not be concerned with such details. Oracle automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity. Oracle also allows the user to lock data manually.

See Also: "Types of Locks" on page 22-20

Transactions and Data Concurrency

Oracle provides data concurrency and integrity between transactions using its locking mechanisms. Because the locking mechanisms of Oracle are tied closely to transaction control, application designers need only define transactions properly, and Oracle automatically manages locking.

Keep in mind that Oracle locking is fully automatic and requires no user action. Implicit locking occurs for all SQL statements so that database users never need to lock any resource explicitly. Oracle's default locking mechanisms lock data at the lowest level of restrictiveness to guarantee data integrity while allowing the highest degree of data concurrency.

See Also: "Explicit (Manual) Data Locking" on page 22-31

Modes of Locking

Oracle uses two modes of locking in a multiuser database:

exclusive lock mode Prevents the associates resource from being shared. This lock mode is obtained to modify 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 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.

Lock Duration

All locks acquired by statements within a transaction are held for the duration of the transaction, preventing destructive interference including dirty reads, lost updates, and destructive DDL operations from concurrent transactions. The

changes made by the SQL statements of one transaction become visible only to other transactions that start *after* the first transaction is committed.

Oracle releases all locks acquired by the statements within a transaction when you either commit or roll back the transaction. Oracle 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 will continue to wait until after the original transaction commits or rolls back completely.

Data Lock Conversion Versus Lock Escalation

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.

Oracle automatically converts a table lock of lower restrictiveness to one of higher restrictiveness as appropriate. For example, assume that a transaction uses a `SELECT` statement with the `FOR UPDATE` clause to lock rows of a table. As a result, it acquires the exclusive row locks and a row share table lock for the table. If the transaction later updates one or more of the locked rows, the row share table lock is automatically converted to a row exclusive table lock.

Lock escalation 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). For example, if a single user locks many rows in a table, some databases automatically escalate the user's row locks to a single table. The number of locks is reduced, but the restrictiveness of what is being locked is increased.

Oracle never escalates locks. Lock escalation greatly increases the likelihood of deadlocks. Imagine the situation where the system is trying to escalate locks on behalf of transaction T1 but cannot because of the locks held by transaction T2. A deadlock is created if transaction T2 also requires lock escalation of the same data before it can proceed.

See Also: "Table Locks (TM)" on page 22-22

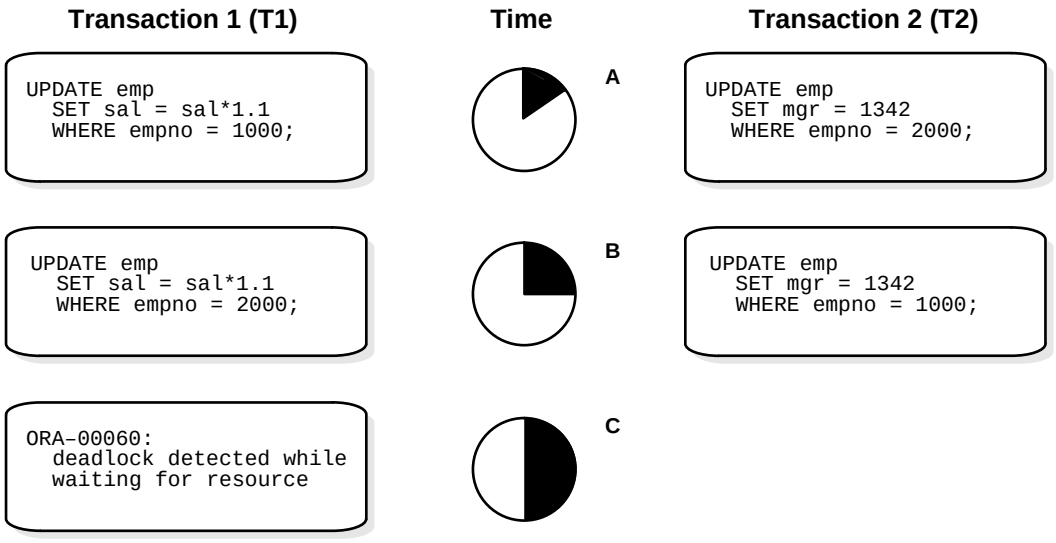
Deadlocks

A **deadlock** can occur when two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work. Figure 22–3 illustrates two transactions in a deadlock.

In Figure 22–3, no problem exists at time point A, as each transaction has a row lock on the row it attempts to update. Each transaction proceeds without being

terminated. However, each tries next to update the row currently held by the other transaction. Therefore, a deadlock results at time point B, because neither transaction can obtain the resource it needs to proceed or terminate. It is a deadlock because no matter how long each transaction waits, the conflicting locks are held.

Figure 22–3 Two Transactions in a Deadlock



Deadlock Detection

Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of the conflicting row locks. A corresponding message also is returned to the transaction that undergoes statement-level rollback. The statement rolled back is the one belonging 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.

> **Note:** In distributed transactions, local deadlocks are detected by analyzing a "waits for" graph, and global deadlocks are detected by a time-out. Once detected, nondistributed and distributed deadlocks are handled by the database and application in the same way.

Deadlocks most often occur when transactions explicitly override the default locking of Oracle. Because Oracle itself does no lock escalation and does not use read locks for queries, but does use row-level locking (rather than page-level locking), deadlocks occur infrequently in Oracle.

> **See Also:** "Explicit (Manual) Data Locking" on page 22-31 for more information about manually acquiring locks

Avoid Deadlocks

Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order, either through implicit or explicit locks. For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table. If such rules are properly designed and then followed in all applications, deadlocks are very unlikely to occur.

When you know you will require a sequence of locks for one transaction, consider acquiring the most exclusive (least compatible) lock first.

Types of Locks

Oracle automatically uses different types of locks to control concurrent access to data and to prevent destructive interaction between users. Oracle automatically locks a resource on behalf of a transaction to prevent other transactions from doing something also requiring exclusive access to the same resource. The lock is released automatically when some event occurs so that the transaction no longer requires the resource.

Throughout its operation, Oracle automatically acquires different types of locks at different levels of restrictiveness depending on the resource being locked and the operation being performed.

Oracle locks fall into one of the following general categories:

DML locks (data locks)	DML locks protect data. For example, table locks lock entire tables, row locks lock selected rows.
DDL locks (dictionary locks)	DDL locks protect the structure of schema objects—for example, the definitions of tables and views.
Internal locks and latches	Internal locks and latches protect internal database structures such as datafiles. Internal locks and latches are entirely automatic.

The following sections discuss DML locks, DDL locks, and internal locks.

DML Locks

The purpose of a DML (data) lock is to guarantee the integrity of data being accessed concurrently by multiple users. DML locks prevent destructive interference of simultaneous conflicting DML and / or DDL operations. For example, Oracle DML locks guarantee that a specific row in a table can be updated by only one transaction at a time and that a table cannot be dropped if an uncommitted transaction contains an insert into the table.

DML operations can acquire data locks at two different levels: for specific rows and for entire tables.

> **Note:** The acronym in parentheses after each type of lock or lock mode is the abbreviation used in the Locks Monitor of Oracle Enterprise Manager. Oracle Enterprise Manager might display TM for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

Row Locks (TX)

The only DML locks Oracle acquires automatically are row-level locks. There is no limit to the number of row locks held by a statement or transaction, and Oracle does not escalate locks from the row level to a coarser granularity. Row locking provides the finest grain locking possible and so provides the best possible concurrency and throughput.

The combination of multiversion concurrency control and row-level locking means that users contend for data only when accessing the same rows, specifically:

- Readers of data do not wait for writers of the same data rows.

- Writers of data do not wait for readers of the same data rows unless `SELECT ... FOR UPDATE` is used, which specifically requests a lock for the reader.
- Writers only wait for other writers if they attempt to update the same rows at the same time.

> **Note:** Readers of data may have to wait for writers of the same data blocks in some very special cases of pending distributed transactions.

A transaction acquires an exclusive DML lock for each individual row modified by one of the following statements: `INSERT`, `UPDATE`, `DELETE`, and `SELECT` with the `FOR UPDATE` clause.

A modified row is **always** locked exclusively so that other users cannot modify the row until the transaction holding the lock is committed or rolled back. However, if the transaction dies due to instance failure, block-level recovery makes a row available before the entire transaction is recovered. Row locks are always acquired automatically by Oracle as a result of the statements listed previously.

If a transaction obtains a row lock for a row, the transaction also acquires a table lock for the corresponding table. The table lock prevents conflicting DDL operations that would override data changes in a current transaction.

See Also: "DDL Locks" on page 22-29

Table Locks (TM)

A transaction acquires a table lock when a table is modified in the following DML statements: `INSERT`, `UPDATE`, `DELETE`, `SELECT` with the `FOR UPDATE` clause, and `LOCK TABLE`. These DML operations require table locks for two purposes: to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction. Any table lock prevents the acquisition of an exclusive DDL lock on the same table and thereby prevents DDL operations that require such locks. For example, a table cannot be altered or dropped if an uncommitted transaction holds a table lock for it.

A table lock can be held in any of several modes: row share (RS), row exclusive (RX), share (S), share row exclusive (SRX), and exclusive (X). The restrictiveness of a table lock's mode determines the modes in which other table locks on the same table can be obtained and held.

Table 22–3 shows the table lock modes that statements acquire and operations that those locks permit and prohibit.

Table 22–3 Summary of Table Locks

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
SELECT...FROM *table*...	none	Y	Y	Y	Y	Y
INSERT INTO *table* ...	RX	Y	Y	N	N	N
UPDATE *table* ...	RX	Y*	Y*	N	N	N
DELETE FROM *table* ...	RX	Y*	Y*	N	N	N
SELECT ... FROM *table* FOR UPDATE OF ...	RS	Y*	Y*	Y*	Y*	N
LOCK TABLE *table* IN ROW SHARE MODE	RS	Y	Y	Y	Y	N
LOCK TABLE *table* IN ROW EXCLUSIVE MODE	RX	Y	Y	N	N	N
LOCK TABLE *table* IN SHARE MODE	S	Y	N	Y	N	N
LOCK TABLE *table* IN SHARE ROW EXCLUSIVE MODE	SRX	Y	N	N	N	N
LOCK TABLE *table* IN EXCLUSIVE MODE	X	N	N	N	N	N
	RS: row share RX: row exclusive S: share SRX: share row exclusive X: exclusive	*Yes, if no conflicting row locks are held by another transaction. Otherwise, waits occur.				

The following sections explain each mode of table lock, from least restrictive to most restrictive. They also describe the actions that cause the transaction to acquire a table lock in that mode and which actions are permitted and prohibited in other transactions by a lock in that mode.

See Also: "Explicit (Manual) Data Locking" on page 22-31

Row Share Table Locks (RS) A row share table lock (also sometimes 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 table lock is automatically acquired for a *table* when one of the following SQL statements is executed:

```
SELECT ... FROM table ... FOR UPDATE OF ... ;

LOCK TABLE table IN ROW SHARE MODE;
```

A row share table lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.

Permitted Operations: A row share table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, other transactions can obtain simultaneous row share, row exclusive, share, and share row exclusive table locks for the same table.

Prohibited Operations: A row share table lock held by a transaction prevents other transactions from exclusive write access to the same table using only the following statement:

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

Row Exclusive Table Locks (RX) A row exclusive table lock (also called a **subexclusive table lock, SX**) generally indicates that the transaction holding the lock has made one or more updates to rows in the table. A row exclusive table lock is acquired automatically for a *table* modified by the following types of statements:

```
INSERT INTO table ... ;

UPDATE table ... ;

DELETE FROM table ... ;

LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

A row exclusive table lock is slightly more restrictive than a row share table lock.

Permitted Operations: A row exclusive table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, row exclusive table locks allow multiple transactions to obtain simultaneous row exclusive and row share table locks for the same table.

Prohibited Operations: A row exclusive table lock held by a transaction prevents other transactions from manually locking the table for exclusive reading or writing. Therefore, other transactions cannot concurrently lock the table using the following statements:

```
LOCK TABLE table IN SHARE MODE;

LOCK TABLE table IN SHARE EXCLUSIVE MODE;

LOCK TABLE table IN EXCLUSIVE MODE;
```

Share Table Locks (S) A share table lock is acquired automatically for the *table* specified in the following statement:

```
LOCK TABLE table IN SHARE MODE;
```

Permitted Operations: A share table lock held by a transaction allows other transactions only to query the table, to lock specific rows with `SELECT ... FOR UPDATE`, or to execute `LOCK TABLE ... IN SHARE MODE` statements successfully. No updates are allowed by other transactions. Multiple transactions can hold share table locks for the same table concurrently. In this case, no transaction can update the table (even if a transaction holds row locks as the result of a `SELECT` statement with the `FOR UPDATE` clause). Therefore, a transaction that has a share table lock can update the table only if no other transactions also have a share table lock on the same table.

Prohibited Operations: A share table lock held by a transaction prevents other transactions from modifying the same table and from executing the following statements:

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

LOCK TABLE table IN EXCLUSIVE MODE;

LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

Share Row Exclusive Table Locks (SRX) A share row exclusive table lock (also sometimes called a **share-subexclusive table lock, SSX**) is more restrictive than a share table lock. A share row exclusive table lock is acquired for a *table* as follows:

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

Permitted Operations: Only one transaction at a time can acquire a share row exclusive table lock on a given table. A share row exclusive table lock held by a transaction allows other transactions to query or lock specific rows using `SELECT` with the `FOR UPDATE` clause, but not to update the table.

Prohibited Operations: A share row exclusive table lock held by a transaction prevents other transactions from obtaining row exclusive table locks and modifying the same table. A share row exclusive table lock also prohibits other transactions from obtaining share, share row exclusive, and exclusive table locks, which prevents other transactions from executing the following statements:

```
LOCK TABLE table IN SHARE MODE;

LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE;

LOCK TABLE table IN ROW EXCLUSIVE MODE;

LOCK TABLE table IN EXCLUSIVE MODE;
```

Exclusive Table Locks (X) An exclusive table lock is the most restrictive mode of table lock, allowing the transaction that holds the lock exclusive write access to the table. An exclusive table lock is acquired for a *table* as follows:

```
LOCK TABLE table IN EXCLUSIVE MODE;
```

Permitted Operations: Only one transaction can obtain an exclusive table lock for a table. An exclusive table lock permits other transactions only to query the table.

Prohibited Operations: An exclusive table lock held by a transaction prohibits other transactions from performing any type of DML statement or placing any type of lock on the table.

DML Locks Automatically Acquired for DML Statements

The previous sections explained the different types of data locks, the modes in which they can be held, when they can be obtained, when they are obtained, and what they prohibit. The following sections summarize how Oracle automatically locks data on behalf of different DML operations.

Table 22–4 summarizes the information in the following sections.

Table 22–4 Locks Obtained By DML Statements

DML Statement	Row Locks?	Mode of Table Lock
SELECT ... FROM *table*		
INSERT INTO *table* ...	X	RX
UPDATE *table* ...	X	RX
DELETE FROM *table* ...	X	RX
SELECT ... FROM *table* ... FOR UPDATE OF ...	X	RS
LOCK TABLE *table* IN ...		
ROW SHARE MODE		RS
ROW EXCLUSIVE MODE		RX
SHARE MODE		S
SHARE EXCLUSIVE MODE		SRX
EXCLUSIVE MODE		X
	X: exclusive RX: row exclusive	RS: row share S: share SRX: share row exclusive

Default Locking for Queries Queries are the SQL statements least likely to interfere with other SQL statements because they only read data. INSERT, UPDATE, and DELETE statements can have implicit queries as part of the statement. Queries include the following kinds of statements:

```
SELECT

INSERT ... SELECT ... ;

UPDATE ... ;

DELETE ... ;
```

They do **not** include the following statement:

```
SELECT ... FOR UPDATE OF ... ;
```

The following characteristics are true of all queries that do not use the FOR UPDATE clause:

- A query acquires no data locks. Therefore, other transactions can query and update a table being queried, including the specific rows being queried. Because queries lacking `FOR UPDATE` clauses do not acquire any data locks to block other operations, such queries are often referred to in Oracle as **nonblocking queries**.
- A query does not have to wait for any data locks to be released; it can always proceed. (Queries may have to wait for data locks in some very specific cases of pending distributed transactions.)

Default Locking for INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE The locking characteristics of `INSERT`, `UPDATE`, `DELETE`, and `SELECT ... FOR UPDATE` statements are as follows:

- The transaction that contains a DML statement acquires exclusive row locks on the rows modified by the statement. Other transactions cannot update or delete the locked rows until the locking transaction either commits or rolls back.
- The transaction that contains a DML statement does not need to acquire row locks on any rows selected by a subquery or an implicit query, such as a query in a `WHERE` clause. A subquery or implicit query in a DML statement is guaranteed to be consistent as of the start of the query and does not see the effects of the DML statement it is part of.
- A query in a transaction can see the changes made by previous DML statements in the same transaction, but cannot see the changes of other transactions begun after its own transaction.
- In addition to the necessary exclusive row locks, a transaction that contains a DML statement acquires at least a row exclusive table lock on the table that contains the affected rows. If the containing transaction already holds a share, share row exclusive, or exclusive table lock for that table, the row exclusive table lock is not acquired. If the containing transaction already holds a row share table lock, Oracle automatically converts this lock to a row exclusive table lock.

DDL Locks

A data dictionary lock (DDL) protects the definition of a schema object while that object is acted upon or referred to by an ongoing DDL operation. Recall that a DDL statement implicitly commits its transaction. For example, assume that a user creates a procedure. On behalf of the user's single-statement transaction, Oracle automatically acquires DDL locks for all schema objects referenced in the procedure definition. The DDL locks prevent objects referenced in the procedure from being altered or dropped before the procedure compilation is complete.

Oracle acquires a dictionary lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. Only individual schema objects that are modified or referenced are locked during DDL operations. The whole data dictionary is never locked.

DDL locks fall into three categories: exclusive DDL locks, share DDL locks, and breakable parse locks.

Exclusive DDL Locks

Most DDL operations, except for those listed in the next section, "Share DDL Locks", 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, a `DROP TABLE` operation is not allowed to drop a table while an `ALTER TABLE` operation is adding a column to it, and vice versa.

During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the schema object by another operation, the acquisition waits until the older DDL lock is released and then proceeds.

DDL operations also acquire DML locks (data locks) on the schema object to be modified.

Share DDL Locks

Some DDL operations require share DDL locks for a resource to prevent destructive interference with conflicting DDL operations, but allow data concurrency for similar DDL operations. For example, when a `CREATE PROCEDURE` statement is executed, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and therefore acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table. No transaction can alter or drop a referenced table. As a result, a transaction that holds

a share DDL lock is guaranteed that the definition of the referenced schema object will remain constant for the duration of the transaction.

A share DDL lock is acquired on a schema object for DDL statements that include the following statements: `AUDIT, NOAUDIT, COMMENT, CREATE [OR REPLACE] VIEW/ PROCEDURE/PACKAGE/PACKAGE BODY/FUNCTION/ TRIGGER, CREATE SYNONYM,` and `CREATE TABLE` (when the `CLUSTER` parameter is not included).

Breakable Parse Locks

A SQL statement (or PL/SQL program unit) in the shared pool holds a parse lock for each schema object 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 does not disallow any DDL operation and can be broken to allow conflicting DDL operations, hence the name **breakable parse lock**.

A parse lock is acquired during the parse phase of SQL statement execution and held as long as the shared SQL area for that statement remains in the shared pool.

See Also: Chapter 19, "Dependencies Among Schema Objects"

Duration of DDL Locks

The duration of a DDL lock depends on its type. Exclusive and share DDL locks last for the duration of DDL statement execution and automatic commit. A parse lock persists as long as the associated SQL statement remains in the shared pool.

DDL Locks and Clusters

A DDL operation on a cluster acquires exclusive DDL locks on the cluster and on all tables and materialized views in the cluster. A DDL operation on a table or materialized view in a cluster acquires a share lock on the cluster, in addition to a share or exclusive DDL lock on the table or materialized view. The share DDL lock on the cluster prevents another operation from dropping the cluster while the first operation proceeds.

Latches and Internal Locks

Latches and internal locks protect internal database and memory structures. Both are inaccessible to users, because users have no need to control over their occurrence or duration. The following section helps to interpret the Oracle Enterprise Manager or SQL*Plus LOCKS and LATCHES monitors.

Latches

Latches are simple, low-level serialization mechanisms to protect shared data structures in the system global area (SGA). For example, latches protect the list of users currently accessing the database and protect the data structures describing the blocks in the buffer cache. A server or background process acquires a latch for a very short time while manipulating or looking at one of these structures. The implementation of latches is operating system dependent, particularly in regard to whether and how long a process will wait for a latch.

Internal Locks

Internal locks are higher-level, more complex mechanisms than latches and serve a variety of purposes.

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. Exclusive locks are released when the DDL operation is complete.

File and Log Management Locks These locks protect various files. For example, one 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 redo log files. Datafiles 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 Rollback Segment Locks These locks protect tablespaces and rollback segments. For example, all instances accessing a database must agree on whether a tablespace is online or offline. Rollback segments are locked so that only one instance can write to a segment.

Explicit (Manual) Data Locking

Oracle always performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, you can override the Oracle default locking mechanisms. Overriding the default locking is useful in situations such as these:

- Applications require transaction-level read consistency or **repeatable reads**. In other words, queries in them 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.

Oracle's automatic locking can be overridden at two levels:

transaction Transactions that include the following SQL statements override Oracle's default locking:

- The `SET TRANSACTION ISOLATION LEVEL` statement
- The `LOCK TABLE` statement (which locks either a table or, when used with views, the underlying base tables)
- The `SELECT ... FOR UPDATE` statement

Locks acquired by these statements are released after the transaction commits or rolls back.

session A session can set the required transaction isolation level with the `ALTER SESSION` statement.

Note: If Oracle's default locking is overridden at any level, 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: *Oracle9i SQL Reference* for detailed descriptions of the SQL statements `LOCK TABLE` and `SELECT ... FOR UPDATE`

Examples of Concurrency under Explicit Locking

The following illustration shows how Oracle maintains data concurrency, integrity, and consistency when `LOCK TABLE` and `SELECT` with the `FOR UPDATE` clause statements are used.

Note: For brevity, the message text for ORA-00054 ("resource busy and acquire with `NOWAIT` specified") is not included. User-entered text is in **bold**.

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept ** IN ROW SHARE MODE;** Statement processed	1	
	2	**DROP TABLE scott.dept;** DROP TABLE scott.dept * ORA-00054 *(exclusive DDL lock not possible because of T1's table lock)*
	3	**LOCK TABLE scott.dept** ** IN EXCLUSIVE MODE NOWAIT;** ORA-00054
	4	**SELECT LOC** ** FROM scott.dept** ** WHERE deptno = 20** ** FOR UPDATE OF loc;** LOC - - - - - - - DALLAS 1 row selected
UPDATE scott.dept ** SET loc = 'NEW YORK'** ** WHERE deptno = 20;** *(waits because T2 has locked same rows)*	5	
	6	**ROLLBACK;** *(releases row locks)*
1 row processed. **ROLLBACK;**	7	

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept ** IN ROW EXCLUSIVE MODE;** Statement processed.	8	
	9	**LOCK TABLE scott.dept** ** IN EXCLUSIVE MODE** ** NOWAIT;** ORA-00054
	10	**LOCK TABLE scott.dept** ** IN SHARE ROW EXCLUSIVE** ** MODE NOWAIT;** ORA-00054
	11	**LOCK TABLE scott.dept** ** IN SHARE ROW EXCLUSIVE** ** MODE NOWAIT;** ORA-00054
	12	**UPDATE scott.dept** ** SET loc = 'NEW YORK'** ** WHERE deptno = 20;** 1 row processed.
	13	**ROLLBACK;**
SELECT loc ** FROM scott.dept** ** WHERE deptno = 20** ** FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.	14	
	15	**UPDATE scott.dept** ** SET loc = 'NEW YORK'** ** WHERE deptno = 20;** *(waits because T1 has locked same rows)*
ROLLBACK;	16	
	17	1 row processed. *(conflicting locks were released)* **ROLLBACK;**

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept IN SHARE MODE Statement processed	18	
	19	**LOCK TABLE scott.dept IN EXCLUSIVE MODE NOWAIT;** ORA-00054
	20	**LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	21	**LOCK TABLE scott.dept IN SHARE MODE;** Statement processed.
	22	**SELECT loc FROM scott.dept WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	23	**SELECT loc FROM scott.dept WHERE deptno = 20 FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.
	24	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** *(waits because T1 holds conflicting table lock)*
ROLLBACK;	25	
	26	1 row processed. *(conflicting table lock released)* **ROLLBACK;**

Transaction 1	Time Point	Transaction 2
LOCK TABLE scott.dept ** IN SHARE ROW** ** EXCLUSIVE MODE;** Statement processed.	27	
	28	**LOCK TABLE scott.dept** ** IN EXCLUSIVE MODE** ** NOWAIT;** ORA-00054
	29	**LOCK TABLE scott.dept** ** IN SHARE ROW** ** EXCLUSIVE MODE** ** NOWAIT;** ORA-00054
	30	**LOCK TABLE scott.dept** ** IN SHARE MODE NOWAIT;** ORA-00054
	31	**LOCK TABLE scott.dept** ** IN ROW EXCLUSIVE** ** MODE NOWAIT;** ORA-00054
	32	**LOCK TABLE scott.dept** ** IN SHARE MODE NOWAIT;** ORA-00054
	33	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	34	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20** ** FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.

Transaction 1	Time Point	Transaction 2
	35	**UPDATE scott.dept** **SET loc = 'NEW YORK'** **WHERE deptno = 20;** *(waits because T1 holds conflicting table lock)*
UPDATE scott.dept **SET loc = 'NEW YORK'** **WHERE deptno = 20;** *(waits because T2 has locked same rows)*	36	*(deadlock)*
Cancel operation **ROLLBACK;**	37	
	38	1 row processed.
LOCK TABLE scott.dept **IN EXCLUSIVE MODE;**	39	
	40	**LOCK TABLE scott.dept** **IN EXCLUSIVE MODE;** ORA-00054
	41	**LOCK TABLE scott.dept** **IN ROW EXCLUSIVE MODE** **NOWAIT;** ORA-00054
	42	**LOCK TABLE scott.dept** **IN SHARE MODE;** ORA-00054
	43	**LOCK TABLE scott.dept** **IN ROW EXCLUSIVE** **MODE NOWAIT;** ORA-00054
	44	**LOCK TABLE scott.dept** **IN ROW SHARE MODE** **NOWAIT;** ORA-00054

Transaction 1	Time Point	Transaction 2
	45	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	46	**SELECT loc** ** FROM scott.dept** ** WHERE deptno = 20** ** FOR UPDATE OF loc;** *(waits because T1 has conflicting table lock)*
UPDATE scott.dept ** SET deptno = 30** ** WHERE deptno = 20;** 1 row processed.	47	
COMMIT;	48	
	49	0 rows selected. *(T1 released conflicting lock)*
SET TRANSACTION READ ONLY;	50	
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - BOSTON	51	
	52	**UPDATE scott.dept** ** SET loc = 'NEW YORK'** ** WHERE deptno = 10;** 1 row processed.

Transaction 1	Time Point	Transaction 2
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - BOSTON *(T1 does not see uncommitted data)*	53	
	54	**COMMIT;**
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - *(same results seen even after T2 commits)*	55	
COMMIT;	56	
SELECT loc ** FROM scott.dept** ** WHERE deptno = 10;** LOC - - - - - - NEW YORK *(committed data is seen)*	57	

Oracle Lock Management Services

With Oracle Lock Management services, an application developer 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

Because a reserved user lock is the same as an Oracle lock, it has all the Oracle lock functionality including deadlock detection. User locks never conflict with Oracle locks, because they are identified with the prefix `UL.`

The Oracle Lock Management services are available through procedures in the `DBMS_LOCK` package.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about Oracle Lock Management services
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about `DBMS_LOCK`

Flashback Query

Oracle9*i* provides a new feature called Flashback Query, which lets you view and repair historical data. Flashback Query offers the ability to perform queries on the database as of a certain wall clock time or user-specified system commit number (SCN).

This new capability uses Oracle's multiversion read-consistency capabilities to restore data by applying undo as needed. Administrators can now configure undo retention by simply specifying how long undo should be kept in the database. Using Flashback Query capabilities, a user can query the database as it existed this morning, yesterday, or last week. The speed of this operation depends only on the amount of data being queried and the number of changes to the data that need to be backed out.

You set the date and time you want to view. Then any SQL query you execute operates on data as it existed at that time. If you are an authorized user, you can

correct errors and back out the restored data without needing the intervention of an administrator.

Flashback Query Benefits

Application Transparency

Packaged applications like report generation tools that only do queries can run in Flashback Query mode, by using log on triggers. Applications can run transparently without requiring changes to code. All the constraints that the application needs to be satisfied are guaranteed to hold good, because we see a consistent version of the database as of the Flashback Query time.

Application Performance

If an application requires recovery actions, it can do so by saving SCNs and flashing back to those SCNs. This is a lot easier and faster than saving data sets and restoring them later, which would be required if the application were to do explicit versioning. Using Flashback Query, we eliminate the costs of logging that would be incurred by explicit versioning.

Online Operation

Flashback Query is an online operation. Concurrent DMLs and queries from other sessions are permitted, while an object is being queried inside Flashback Query.The speed of these operations is unaffected. Moreover, different sessions can flash back to different flashback times or SCNs on the same object, concurrently. The speed of the Flashback Query itself depends on the amount of undo that needs to be applied proportional to how far back in time the query goes.

Easy Manageability

No additional management on the part of the user, except setting the appropriate retention interval, having the right privileges etc. No additional logging has to be turned on because past versions are constructed automatically, as needed.

Notes:

- Flashback Query does NOT undo anything. It is only a query mechanism. You can take the output from a flashback query and perform an Undo yourself in many circumstances.
- Flashback Query does NOT tell you what changed. LogMiner does that.
- Flashback Query can be used to undo changes and can be very efficient if you know the rows that need to be moved back in time. You can in theory use it to move a full table back in time but this is very expensive if the table is large since it involves a full table copy.
- Flashback Query does not work through DDL operations that modify columns, or drop or truncate tables.
- LogMiner is very good for getting change history, but it gives you changes in terms of deltas (insert, update, delete), not in terms of the before and after image of a row. These can be difficult to deal with in some applications.

Some Uses of Flashback Query

Self-Service Repair

Perhaps you accidentally deleted some important rows from a table and wanted to recover the deleted rows. To do the repair, you can move backwards in time and see the missing rows and re-insert the deleted row into the current table.

E-Mail or Voice Mail Applications

You might have deleted mail in the past. Using Flashback Query, you can restore the deleted mail by moving back in time and re-inserting the deleted message into the current message box.

Account Balances

You can view account prior account balances as of a certain day in the month.

DSS and OLAP Applications

Decision Support Systems (DSS) and Online Analytical Processing (OLAP) applications must perform long-running transactions. With Flashback Query, these applications can perform data analysis and modeling on historical data. For example, seasonal demand for a particular product could be reviewed.

Packaged Applications

Packaged Applications (like report generation tools) can make use of Flashback Query without any changes to application logic. Any constraints that the application expects are guaranteed to be satisfied, because users see a consistent version of the Database as of the given time or SCN.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about using Flashback Query
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for a description of the DBMS_FLASHBACK package
- *Oracle9i Database Administrator's Guide* for information about undo tablespaces and setting retention period

23

Data Integrity

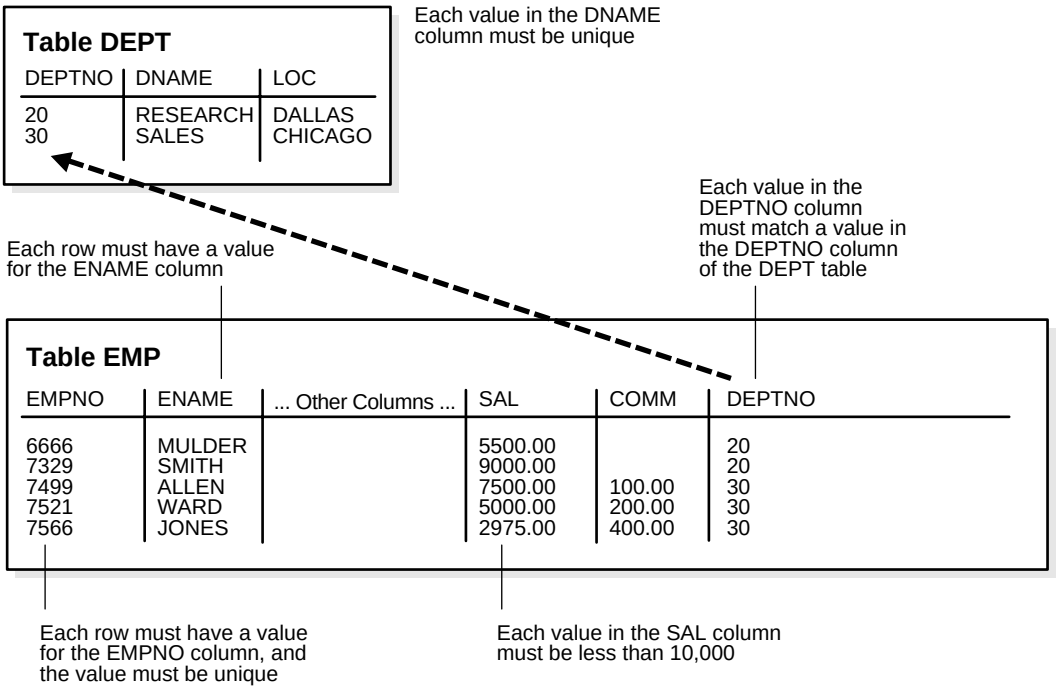
This chapter explains how to use integrity constraints to enforce the business rules associated with your database and prevent the entry of invalid information into tables. The chapter includes:

- Introduction to Data Integrity
- Introduction to Integrity Constraints
- Types of Integrity Constraints
- The Mechanisms of Constraint Checking
- Deferred Constraint Checking
- Constraint States

Introduction to Data Integrity

It is important that data adhere to a predefined set of rules, as determined by the database administrator or application developer. As an example of data integrity, consider the tables EMP and DEPT and the business rules for the information in each of the tables, as illustrated in Figure 23–1.

Figure 23–1 Examples of Data Integrity



Note that some columns in each table have specific rules that constrain the data contained within them.

Types of Data Integrity

This section describes the rules that can be applied to table columns to enforce different types of data integrity.

Null Rule

A null is a rule defined on a single column that allows or disallows inserts or updates of rows containing a null (the absence of a value) in that column.

Unique Column Values

A unique value defined on a column (or set of columns) allows the insert or update of a row only if it contains a unique value in that column (or set of columns).

Primary Key Values

A primary key value defined on a key (a column or set of columns) specifies that each row in the table can be uniquely identified by the values in the key.

Referential Integrity Rules

A rule defined on a key (a column or set of columns) in one table that guarantees that the values in that key match the values in a key in a related table (the referenced value).

Referential integrity also includes the rules that dictate what types of data manipulation are allowed on referenced values and how these actions affect dependent values. The rules associated with referential integrity are:

Restrict	Disallows the update or deletion of referenced data.
Set to Null	When referenced data is updated or deleted, all associated dependent data is set to `NULL`.
Set to Default	When referenced data is updated or deleted, all associated dependent data is set to a default value.
Cascade	When referenced data is updated, all associated dependent data is correspondingly updated. When a referenced row is deleted, all associated dependent rows are deleted.
No Action	Disallows the update or deletion of referenced data. This differs from `RESTRICT` in that it is checked at the end of the statement, or at the end of the transaction if the constraint is deferred. (Oracle uses No Action as its default action.)

Complex Integrity Checking

Complex integrity checking is a user-defined rule for a column (or set of columns) that allows or disallows inserts, updates, or deletes of a row based on the value it contains for the column (or set of columns).

How Oracle Enforces Data Integrity

Oracle enables you to define and enforce each type of data integrity rule defined in the previous section. Most of these rules are easily defined using integrity constraints or database triggers.

Integrity Constraints Description

An integrity constraint is a declarative method of defining a rule for a column of a table. Oracle supports the following integrity constraints:

- `NOT NULL` constraints for the rules associated with nulls in a column
- `UNIQUE key` constraints for the rule associated with unique column values
- `PRIMARY KEY` constraints for the rule associated with primary identification values
- `FOREIGN KEY` constraints for the rules associated with referential integrity. Oracle supports the use of `FOREIGN KEY` integrity constraints to define the referential integrity actions, including:
  - Update and delete No Action
  - Delete `CASCADE`
  - Delete `SET NULL`
- `CHECK` constraints for complex integrity rules

Note: You cannot enforce referential integrity using declarative integrity constraints if child and parent tables are on different nodes of a distributed database. However, you can enforce referential integrity in a distributed database using database triggers (see next section).

Database Triggers

Oracle also allows you to enforce integrity rules with a non-declarative approach using database triggers (stored database procedures automatically invoked on insert, update, or delete operations).

> **See Also:** Chapter 18, "Triggers" for examples of triggers used to enforce data integrity

Introduction to Integrity Constraints

Oracle uses integrity constraints to prevent invalid data entry into the base tables of the database. You can define integrity constraints to enforce the business rules you want to associate with the information in a database. If any of the results of a DML statement execution violate an integrity constraint, then Oracle rolls back the statement and returns an error.

> **Note:** Operations on views (and synonyms for tables) are subject to the integrity constraints defined on the underlying base tables.

For example, assume that you define an integrity constraint for the `SAL` column of the `EMP` table. This integrity constraint enforces the rule that no row in this table can contain a numeric value greater than 10,000 in this column. If an `INSERT` or `UPDATE` statement attempts to violate this integrity constraint, then Oracle rolls back the statement and returns an information error message.

The integrity constraints implemented in Oracle fully comply with ANSI X3.135-1989 and ISO 9075-1989 standards.

Advantages of Integrity Constraints

This section describes some of the advantages that integrity constraints have over other alternatives, which include:

- Enforcing business rules in the code of a database application
- Using stored procedures to completely control access to data
- Enforcing business rules with triggered stored database procedures

> **See Also:** Chapter 18, "Triggers"

Declarative Ease

Define integrity constraints using SQL statements. When you define or alter a table, no additional programming is required. The SQL statements are easy to write and eliminate programming errors. Oracle controls their functionality. For these reasons, declarative integrity constraints are preferable to application code and database triggers. The declarative approach is also better than using stored procedures, because the stored procedure solution to data integrity controls data access, but integrity constraints do not eliminate the flexibility of ad hoc data access.

Centralized Rules

Integrity constraints are defined for tables (not an application) and are stored in the data dictionary. Any data entered by any application must adhere to the same integrity constraints associated with the table. By moving business rules from application code to centralized integrity constraints, the tables of a database are guaranteed to contain valid data, no matter which database application manipulates the information. Stored procedures cannot provide the same advantage of centralized rules stored with a table. Database triggers can provide this benefit, but the complexity of implementation is far greater than the declarative approach used for integrity constraints.

Maximum Application Development Productivity

If a business rule enforced by an integrity constraint changes, then the administrator need only change that integrity constraint and all applications automatically adhere to the modified constraint. In contrast, if the business rule were enforced by the code of each database application, developers would have to modify all application source code and recompile, debug, and test the modified applications.

Immediate User Feedback

Oracle stores specific information about each integrity constraint in the data dictionary. You can design database applications to use this information to provide immediate user feedback about integrity constraint violations, even before Oracle executes and checks the SQL statement. For example, a SQL*Forms application can use integrity constraint definitions stored in the data dictionary to check for violations as values are entered into the fields of a form, even before the application issues a statement.

Superior Performance

The semantics of integrity constraint declarations are clearly defined, and performance optimizations are implemented for each specific declarative rule. The

Oracle query optimizer can use declarations to learn more about data to improve overall query performance. (Also, taking integrity rules out of application code and database triggers guarantees that checks are only made when necessary.)

Flexibility for Data Loads and Identification of Integrity Violations

You can disable integrity constraints temporarily so that large amounts of data can be loaded without the overhead of constraint checking. When the data load is complete, you can easily enable the integrity constraints, and you can automatically report any new rows that violate integrity constraints to a separate exceptions table.

The Performance Cost of Integrity Constraints

The advantages of enforcing data integrity rules come with some loss in performance. In general, the cost of including an integrity constraint is, at most, the same as executing a SQL statement that evaluates the constraint.

Types of Integrity Constraints

You can use the following integrity constraints to impose restrictions on the input of column values:

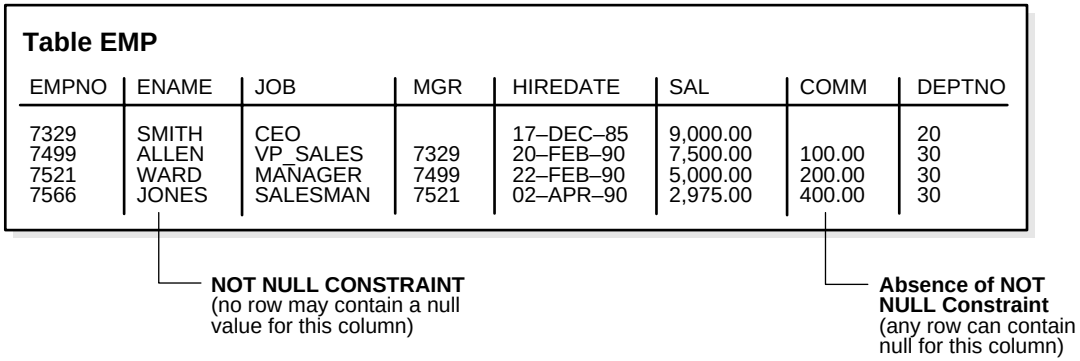
- NOT NULL Integrity Constraints
- UNIQUE Key Integrity Constraints
- PRIMARY KEY Integrity Constraints
- Referential Integrity Constraints
- CHECK Integrity Constraints

NOT NULL Integrity Constraints

By default, all columns in a table allow nulls. **Null** means the absence of a value. A `NOT NULL` constraint requires a column of a table contain no null values. For example, you can define a `NOT NULL` constraint to require that a value be input in the `ENAME` column for every row of the `EMP` table.

Figure 23–2 illustrates a `NOT NULL` integrity constraint.

Figure 23–2 NOT NULL Integrity Constraints

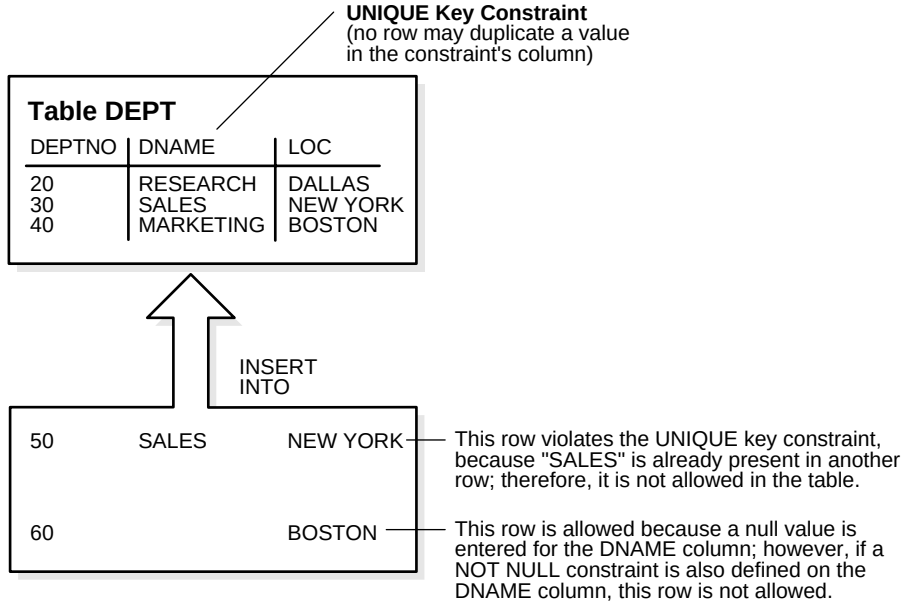


UNIQUE Key Integrity Constraints

A `UNIQUE` key integrity constraint requires that every value in a column or set of columns (key) be unique—that is, no two rows of a table have duplicate values in a specified column or set of columns.

For example, in Figure 23–3 a `UNIQUE` key constraint is defined on the `DNAME` column of the `DEPT` table to disallow rows with duplicate department names.

Figure 23–3 A UNIQUE Key Constraint

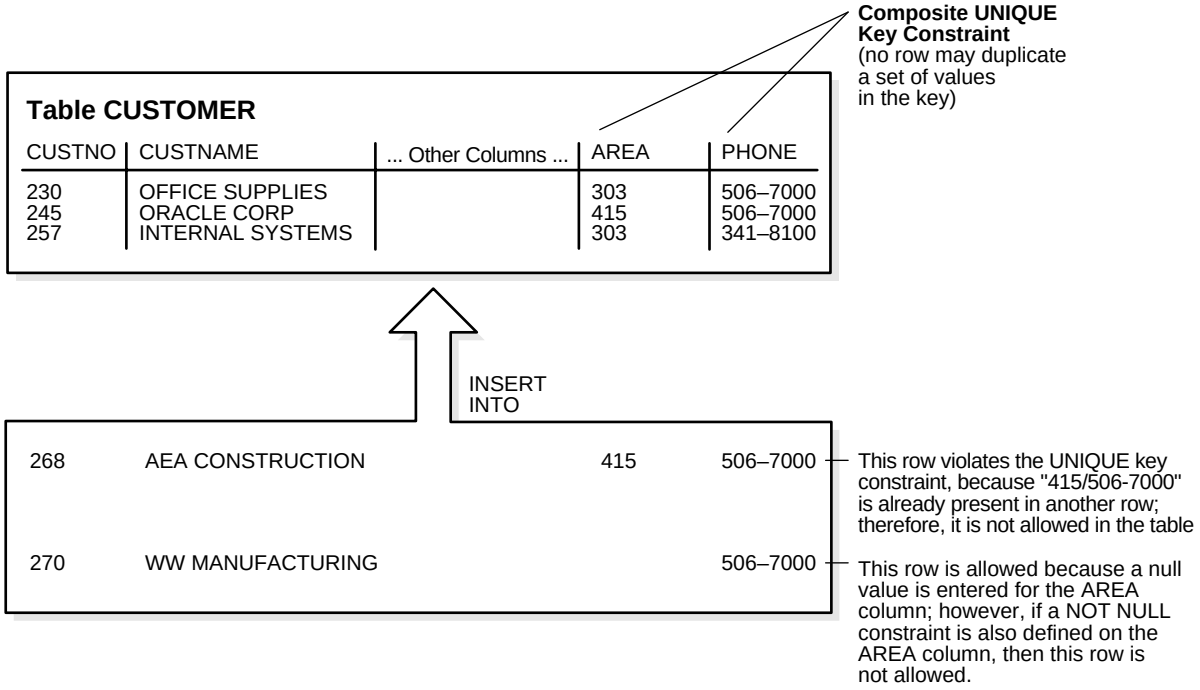


Unique Keys

The columns included in the definition of the `UNIQUE` key constraint are called the **unique key**. **Unique key** is often incorrectly used as a synonym for the terms **UNIQUE key constraint** or **UNIQUE index**. However, note that **key** refers only to the column or set of columns used in the definition of the integrity constraint.

If the `UNIQUE` key consists of more than one column, that group of columns is said to be a **composite unique key**. For example, in Figure 23–4 the `CUSTOMER` table has a `UNIQUE` key constraint defined on the composite unique key: the `AREA` and `PHONE` columns.

Figure 23–4 A Composite UNIQUE Key Constraint



This `UNIQUE` key constraint allows you to enter an area code and telephone number any number of times, but the combination of a given area code and given telephone number cannot be duplicated in the table. This eliminates unintentional duplication of a telephone number.

UNIQUE Key Constraints and Indexes

Oracle enforces unique integrity constraints with indexes. For example, in Figure 23–4, Oracle enforces the `UNIQUE` key constraint by implicitly creating a unique index on the composite unique key. Therefore, composite `UNIQUE` key constraints have the same limitations imposed on composite indexes: up to 32 columns can constitute a composite unique key.

> **Note:** If compatibility is set to Oracle9*i*, Release 1 (9.0.1) or higher, the total size in bytes of a key value can be almost as large as a full block. In previous releases key size could not exceed approximately half the associated database's block size.

If a usable index exists when a unique key constraint is created, the constraint uses that index rather than implicitly creating a new one.

Combine UNIQUE Key and NOT NULL Integrity Constraints

In Figure 23–3 and Figure 23–4, `UNIQUE` key constraints allow the input of nulls unless you also define `NOT NULL` constraints for the same columns. In fact, any number of rows can include nulls for columns without `NOT NULL` constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite `UNIQUE` key) always satisfies a `UNIQUE` key constraint.

Columns with both unique keys and `NOT NULL` integrity constraints are common. This combination forces the user to enter values in the unique key and also eliminates the possibility that any new row's data will ever conflict with an existing row's data.

> **Note:** Because of the search mechanism for `UNIQUE` constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite `UNIQUE` key constraint.

PRIMARY KEY Integrity Constraints

Each table in the database can have at most one `PRIMARY KEY` constraint. The values in the group of one or more columns subject to this constraint constitute the unique identifier of the row. In effect, each row is named by its primary key values.

The Oracle implementation of the `PRIMARY KEY` integrity constraint guarantees that both of the following are true:

- No two rows of a table have duplicate values in the specified column or set of columns.
- The primary key columns do not allow nulls. That is, a value must exist for the primary key columns in each row.

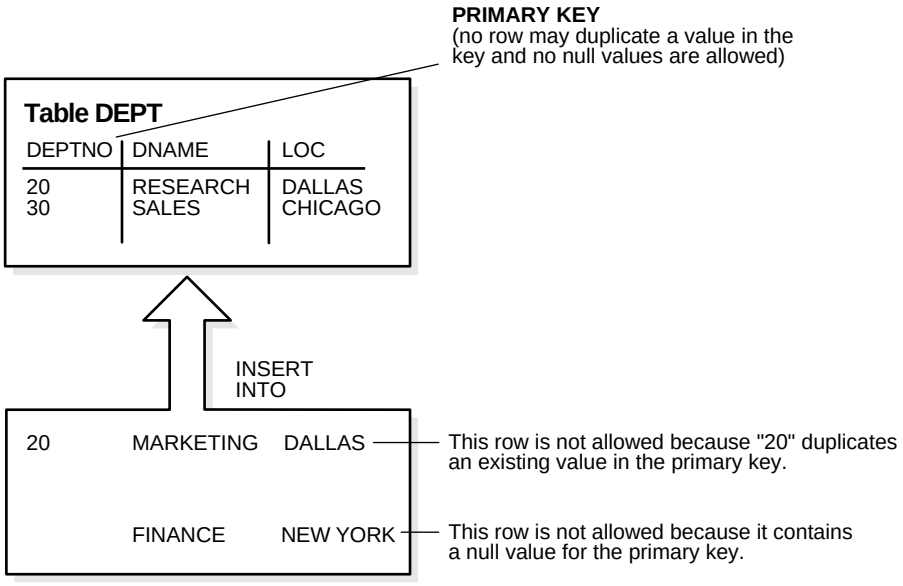
Primary Keys

The columns included in the definition of a table's `PRIMARY KEY` integrity constraint are called the *primary key*. Although it is not required, every table should have a primary key so that:

- Each row in the table can be uniquely identified
- No duplicate rows exist in the table

Figure 23–5 illustrates a `PRIMARY KEY` constraint in the `DEPT` table and examples of rows that violate the constraint.

Figure 23–5 A Primary Key Constraint



PRIMARY KEY Constraints and Indexes

Oracle enforces all `PRIMARY KEY` constraints using indexes. In Figure 23–5, the primary key constraint created for the `DEPTNO` column is enforced by the implicit creation of:

- A unique index on that column
- A `NOT NULL` constraint for that column

Oracle enforces primary key constraints using indexes, and composite primary key constraints are limited to 32 columns, which is the same limitation imposed on composite indexes. The name of the index is the same as the name of the constraint. Also, you can specify the storage options for the index by including the `ENABLE` clause in the `CREATE TABLE` or `ALTER TABLE` statement used to create the constraint. If a usable index exists when a primary key constraint is created, then the primary key constraint uses that index rather than implicitly creating a new one.

Referential Integrity Constraints

Different tables in a relational database can be related by common columns, and the rules that govern the relationship of the columns must be maintained. Referential integrity rules guarantee that these relationships are preserved.

Several terms are associated with referential integrity constraints:

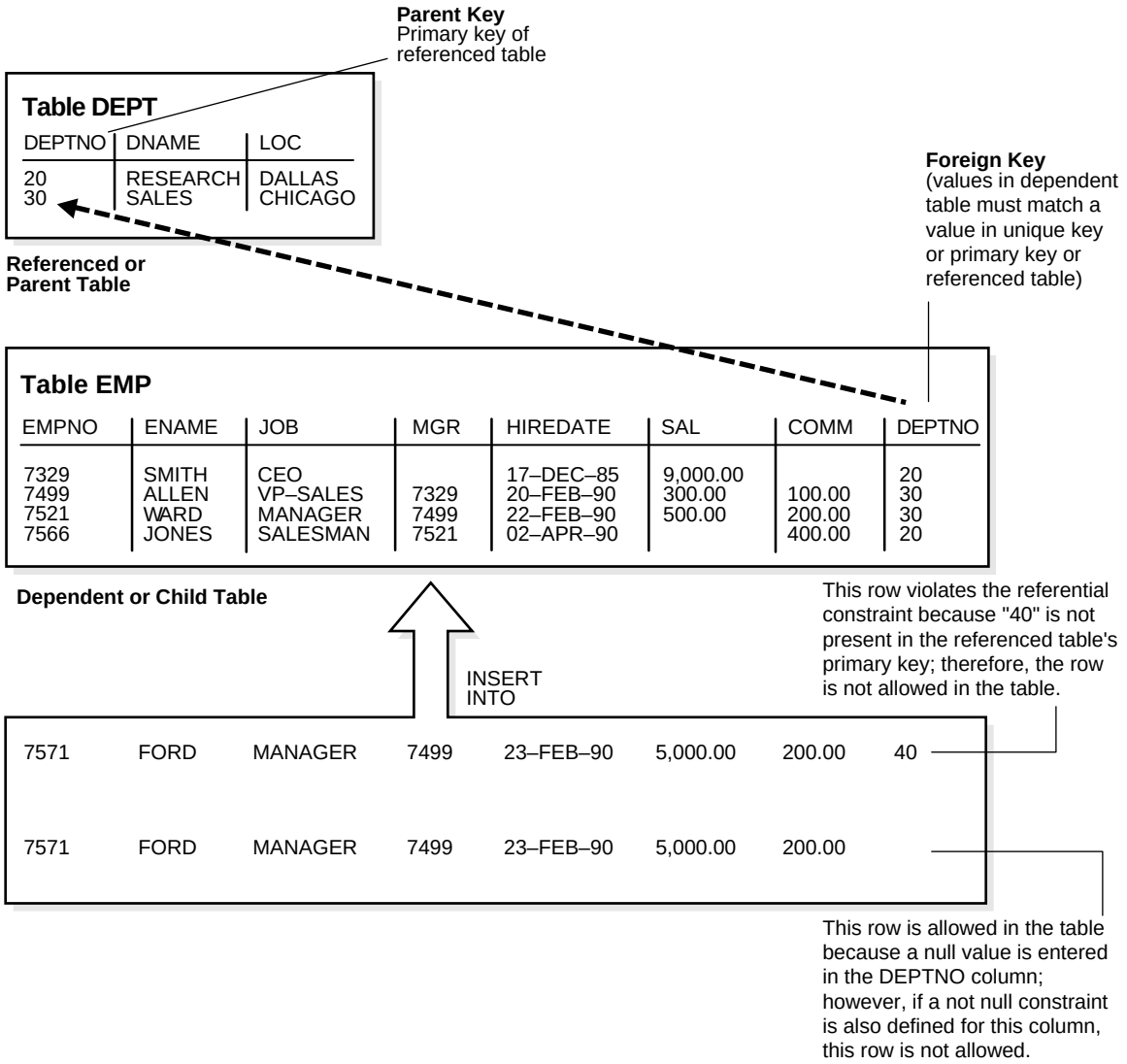
Foreign key	The column or set of columns included in the definition of the referential integrity constraint that reference a referenced key.
Referenced key	The unique key or primary key of the same or different table that is referenced by a foreign key.
Dependent or **child table**	The table that includes the foreign key. Therefore, it is the table that is dependent on the values present in the referenced unique or primary key.
Referenced or **parent table**	The table that is referenced by the child table's foreign key. It is this table's referenced key that determines whether specific inserts or updates are allowed in the child table.

A referential integrity constraint requires that for each row of a table, the value in the foreign key matches a value in a parent key.

Figure 23–6 shows a foreign key defined on the `DEPTNO` column of the `EMP` table. It guarantees that every value in this column must match a value in the primary key of the `DEPT` table (also the `DEPTNO` column). Therefore, no erroneous department numbers can exist in the `DEPTNO` column of the `EMP` table.

Foreign keys can 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 datatypes. Because composite primary and unique keys are limited to 32 columns, a composite foreign key is also limited to 32 columns.

Figure 23–6 Referential Integrity Constraints



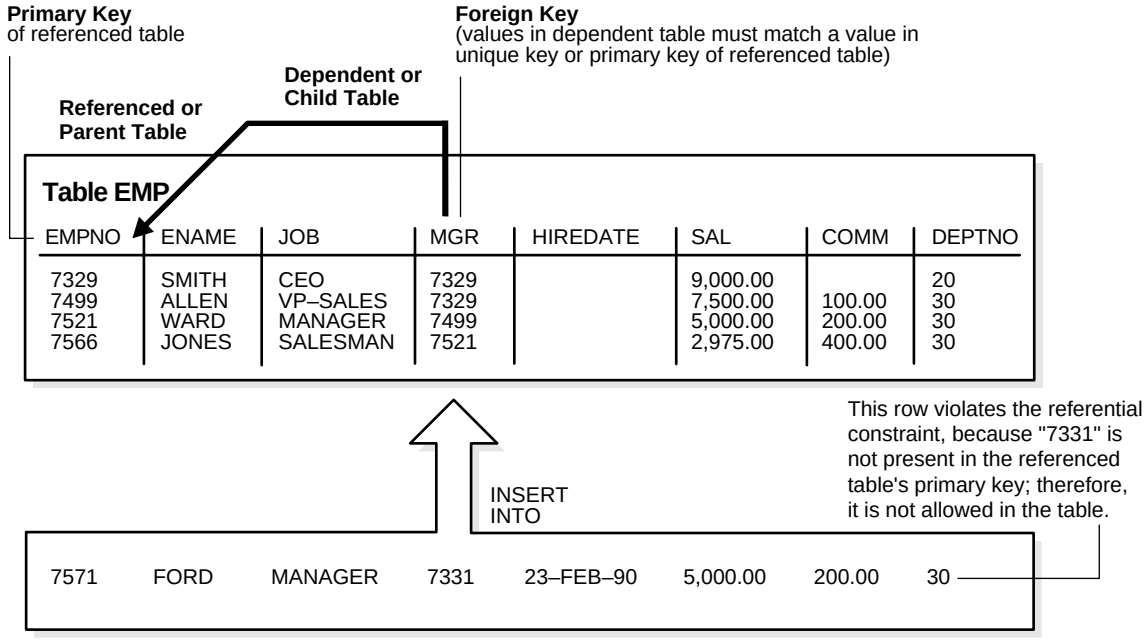
Self-Referential Integrity Constraints

Another type of referential integrity constraint, shown in Figure 23–7, is called a

self-referential integrity constraint. This type of foreign key references a parent key in the same table.

In Figure 23–7, the referential integrity constraint ensures that every value in the MGR column of the EMP table corresponds to a value that currently exists in the EMPNO column of the same table, but not necessarily in the same row, because every manager must also be an employee. This integrity constraint eliminates the possibility of erroneous employee numbers in the MGR column.

Figure 23–7 Single Table Referential Constraints



Nulls and Foreign Keys

The relational model permits the value of foreign keys either to match 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.

Actions Defined by Referential Integrity Constraints

Referential integrity constraints can specify particular actions to be performed on the dependent rows in a child table if a referenced parent key value is modified. The referential actions supported by the FOREIGN KEY integrity constraints of Oracle are UPDATE and DELETE No Action, and DELETE CASCADE.

Note: Other referential actions not supported by FOREIGN KEY integrity constraints of Oracle can be enforced using database triggers.

See Chapter 18, "Triggers" for more information.

Update and Delete No Action The No Action (default) option specifies that referenced key values cannot be updated or deleted if the resulting data would violate a referential integrity constraint. For example, if a primary key value is referenced by a value in the foreign key, then the referenced primary key value cannot be deleted because of the dependent data.

Delete Cascade A **delete cascades** 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, if a row in a parent table is deleted, and this row's primary key value is referenced by one or more foreign key values in a child table, then the rows in the child table that reference the primary key value are also deleted from the child table.

Delete Set Null A delete **sets 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, if EMPNO references MGR in the TMP table, then deleting a manager causes the rows for all employees working for that manager to have their MGR value set to null.

DML Restrictions with Respect to Referential Actions Table 23–1 outlines the DML statements allowed by the different referential actions on the primary / unique key values in the parent table, and the foreign key values in the child table.

Table 23–1 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.

Concurrency Control, Indexes, and Foreign Keys

You almost always index foreign keys. The only exception is when the matching unique or primary key is never updated or deleted.

Oracle maximizes the concurrency control of parent keys in relation to dependent foreign key values. You can control what concurrency mechanisms are used to maintain these relationships, and, depending on the situation, this can be highly beneficial. The following sections explain the possible situations and give recommendations for each.

No Index on the Foreign Key Figure 23–8 illustrates the locking mechanisms used by Oracle when no index is defined on the foreign key and when rows are being updated or deleted in the parent table. Inserts into the parent table do not require any locks on the child table.

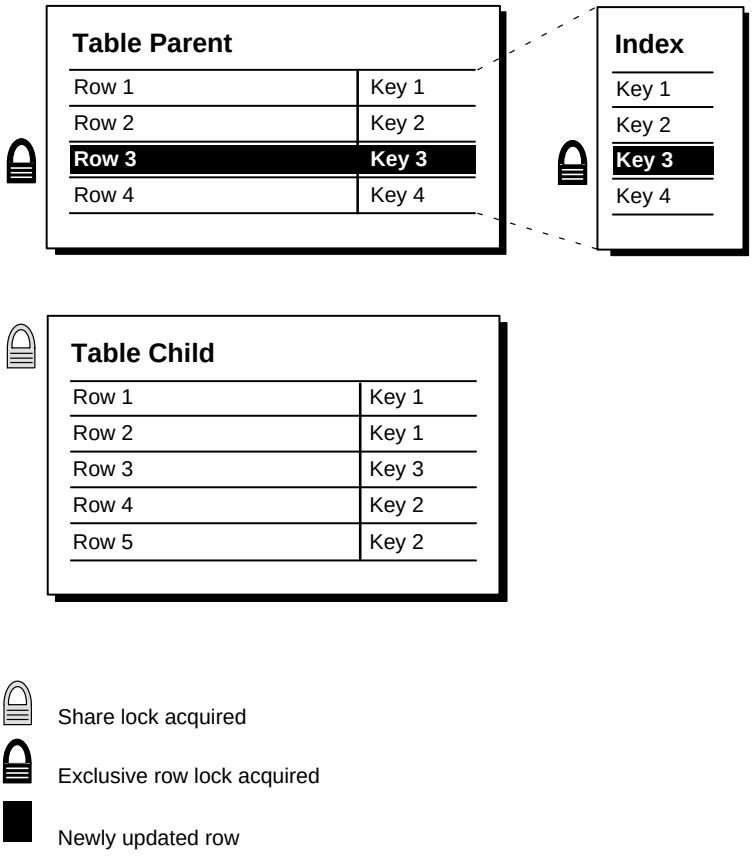
Oracle9*i*, Release 1 (9.0.1), no longer requires a share lock on unindexed foreign keys when doing an update or delete on the primary key. It still obtains the table-level share lock, but then releases it immediately after obtaining it. If multiple primary keys are update or deleted, the lock is obtained and released once for each row.

In previous releases, a share lock of the entire child table was required until the transaction containing the `DELETE` statement for the parent table was committed.

If the foreign key specifies `ON DELETE CASCADE`, then the `DELETE` statement resulted in a table-level share-subexclusive lock on the child table. A share lock of the entire child table was also required for an `UPDATE` statement on the parent table that affected any columns referenced by the child table. Share locks allow reading only. Therefore, no `INSERT`, `UPDATE`, or `DELETE` statements could be issued on the child table until the transaction containing the `UPDATE` or `DELETE` was committed. Queries were allowed on the child table.

`INSERT`, `UPDATE`, and `DELETE` statements on the child table do not acquire any locks on the parent table, although `INSERT` and `UPDATE` statements wait for a row-lock on the index of the parent table to clear.

Figure 23–8 Locking Mechanisms When No Index Is Defined on the Foreign Key

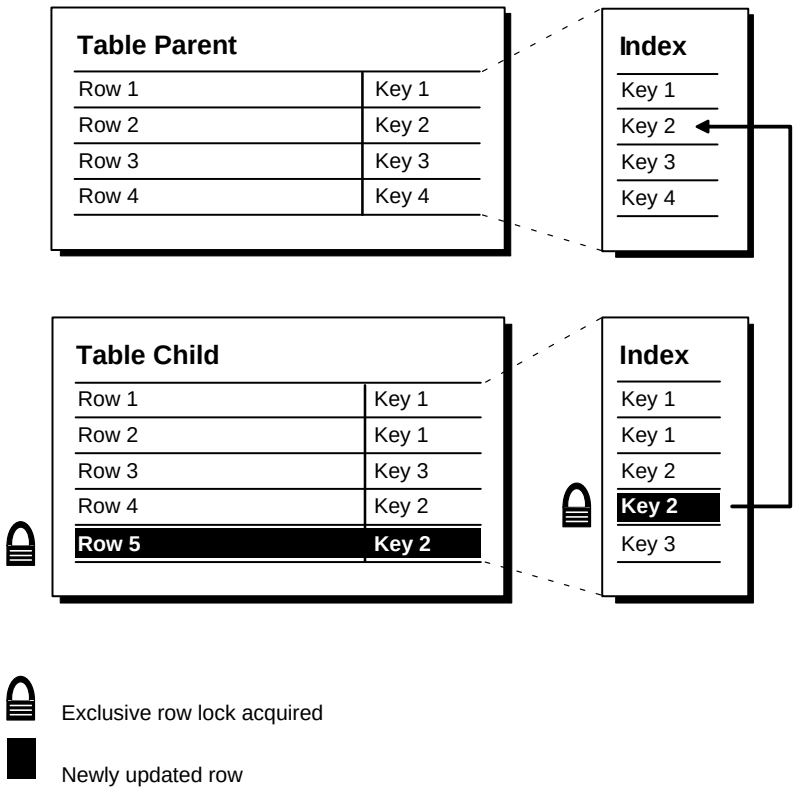


Index on the Foreign Key Figure 23–9 illustrates the locking mechanisms used by Oracle when an index is defined on the foreign key, and new rows are inserted, updated, or deleted in the child table.

Notice that no table locks of any kind are acquired on the parent table or any of its indexes as a result of the insert, update, or delete. Therefore, any type of DML statement can be issued on the parent table, including inserts, updates, deletes, and queries.

This situation is preferable if there is any update or delete activity on the parent table while update activity is taking place on the child table. Inserts, updates, and deletes on the parent table do not require any locks on the child table, although updates and deletes will wait for row-level locks on the indexes of the child table to clear.

Figure 23–9 Locking Mechanisms When Index Is Defined on the Foreign Key



If the child table specifies `ON DELETE CASCADE`, then deletes from the parent table can result in deletes from the child table. In this case, waiting and locking rules are the same as if you deleted yourself from the child table after performing the delete from the parent table.

CHECK Integrity Constraints

A CHECK integrity constraint on a column or set of columns requires that a specified condition be true or unknown for every row of the table. If a DML statement results in the condition of the CHECK constraint evaluating to false, then the statement is rolled back.

The Check Condition

CHECK constraints enable you to enforce very specific integrity rules by specifying a check condition. The condition of a CHECK constraint has some limitations:

- It must be a Boolean expression evaluated using the values in the row being inserted or updated, and
- It cannot contain subqueries; sequences; the SQL functions SYSDATE, UID, USER, or USERENV; or the pseudocolumns LEVEL or ROWNUM.

In evaluating CHECK constraints that contain string literals or SQL functions with Globalization Support parameters as arguments (such as TO_CHAR, TO_DATE, and TO_NUMBER), Oracle uses the database's Globalization Support settings by default. You can override the defaults by specifying Globalization Support parameters explicitly in such functions within the CHECK constraint definition.

> **See Also:** *Oracle9i Globalization Support Guide* for more information on Globalization Support features

Multiple CHECK Constraints

A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

If you create multiple CHECK constraints for a column, design them carefully so their purposes do not conflict. Do not assume any particular order of evaluation of the conditions. Oracle does not verify that CHECK conditions are not mutually exclusive.

The Mechanisms of Constraint Checking

To know what types of actions are permitted when constraints are present, it is useful to understand when Oracle actually performs the checking of constraints. To illustrate this, an example or two is helpful. Assume the following:

- The EMP table has been defined as in Figure 23–7 on page 23-15.

- The self-referential constraint makes the entries in the MGR column dependent on the values of the EMPNO column. For simplicity, the rest of this discussion addresses only the EMPNO and MGR columns of the EMP table.

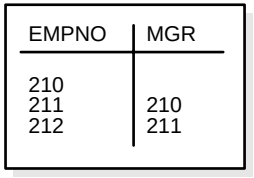
Consider the insertion of the first row into the EMP table. No rows currently exist, so how can a row be entered if the value in the MGR column cannot reference any existing value in the EMPNO column? Three possibilities for doing this are:

- A null can be entered for the MGR column of the first row, assuming that the MGR column does not have a NOT NULL constraint defined on it. Because nulls are allowed in foreign keys, this row is inserted successfully into the table.
- The same value can be entered in both the EMPNO and MGR columns. This case reveals that Oracle performs its constraint checking *after* the statement has been completely executed. To allow a row to be entered with the same values in the parent key and the foreign key, Oracle must first execute the statement (that is, insert the new row) and then check to see if any row in the table has an EMPNO that corresponds to the new row's MGR.
- 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 EMPNO as 200 and MGR as 300, while the second row might have EMPNO as 300 and MGR as 200.

  This case also shows that constraint checking is deferred until the complete execution of the statement. All rows are inserted first, then all rows are checked for constraint violations. You can also defer the checking of constraints until the end of the **transaction**.

Consider the same self-referential integrity constraint in this 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 new company's employee numbers. Because manager numbers are really employee numbers, these values must also increase by 5000 (see Figure 23–10).

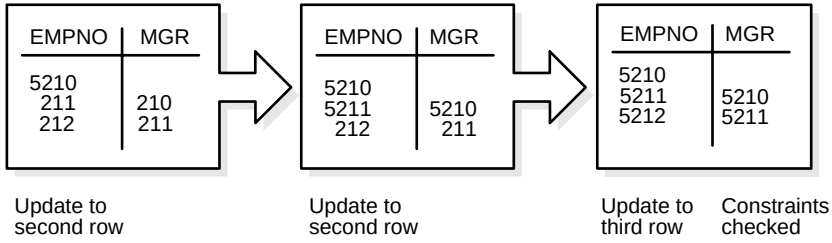
Figure 23–10 The EMP Table Before Updates




```
UPDATE emp
  SET empno = empno + 5000,
      mgr = mgr + 5000;
```

Even though a constraint is defined to verify that each `MGR` value matches an `EMPNO` value, this statement is legal because Oracle effectively performs its constraint checking after the statement completes. Figure 23–11 shows that Oracle performs the actions of the entire SQL statement before any constraints are checked.

Figure 23–11 Constraint Checking



The examples in this section illustrate the constraint checking mechanism during `INSERT` and `UPDATE` statements. The same mechanism is used for all types of DML statements, including `UPDATE`, `INSERT`, and `DELETE` statements.

The examples also used self-referential integrity constraints to illustrate the checking mechanism. The same mechanism is used for all types of constraints, including the following:

- `NOT NULL`
- `UNIQUE` key
- `PRIMARY KEY`
- all types of `FOREIGN KEY` constraints

- `CHECK` constraints

See Also: "Deferred Constraint Checking" on page 23-24

Default Column Values and Integrity Constraint Checking

Default values are included as part of an `INSERT` statement before the statement is parsed. Therefore, default column values are subject to all integrity constraint checking.

Deferred Constraint Checking

You can **defer** checking constraints for validity until the end of the transaction.

- A constraint is **deferred** if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back.
- If a constraint is **immediate** (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately.

If a constraint causes an **action** (for example, delete cascade), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate.

Constraint Attributes

You can define constraints as either **deferrable** or **not deferrable**, and either **initially deferred** or **initially immediate**. These attributes can be different for each constraint. You specify them with keywords in the `CONSTRAINT` clause:

- `DEFERRABLE or NOT DEFERRABLE`
- `INITIALLY DEFERRED or INITIALLY IMMEDIATE`

Constraints can be added, dropped, enabled, disabled, or validated. You can also modify a constraint's attributes.

See Also:

- *Oracle9i SQL Reference* for information about constraint attributes and their default values
- "Constraint States" on page 23-26
- "Constraint State Modification" on page 23-27

SET CONSTRAINTS Mode

The SET CONSTRAINTS statement makes constraints either DEFERRED or IMMEDIATE for a particular transaction (following the ANSI SQL92 standards in both syntax and semantics). You can use this statement to set the mode for a list of constraint names or for ALL constraints.

The SET CONSTRAINTS mode lasts for the duration of the transaction or until another SET CONSTRAINTS statement resets the mode.

SET CONSTRAINTS ... IMMEDIATE causes the specified constraints to be checked immediately on execution of each constrained statement. Oracle first checks any constraints that were deferred earlier in the transaction and then continues immediately checking constraints of any further statements in that transaction, as long as all the checked constraints are consistent and no other SET CONSTRAINTS statement is issued. If any constraint fails the check, an error is signaled. At that point, a COMMIT causes the whole transaction to roll back.

The ALTER SESSION statement also has clauses to SET CONSTRAINTS IMMEDIATE or DEFERRED. These clauses imply setting ALL deferrable constraints (that is, you cannot specify a list of constraint names). They are equivalent to making a SET CONSTRAINTS statement at the start of each transaction in the current session.

Making constraints **immediate** at the end of a transaction is a way of checking whether COMMIT can succeed. You can avoid unexpected rollbacks by setting constraints to IMMEDIATE as the last statement in a transaction. If any constraint fails the check, you can then correct the error before committing the transaction.

The SET CONSTRAINTS statement is disallowed inside of triggers.

SET CONSTRAINTS can be a distributed statement. Existing database links that have transactions in process are told when a SET CONSTRAINTS ALL statement occurs, and new links learn that it occurred as soon as they start a transaction.

Unique Constraints and Indexes

A user sees inconsistent constraints, including duplicates in unique indexes, when that user's transaction produces these inconsistencies.

You can place deferred unique and foreign key constraints on materialized views, allowing fast and complete refresh to complete successfully.

Deferrable unique constraints always use nonunique indexes. When you remove a deferrable constraint, its index remains. This is convenient because the storage information remains available after you disable a constraint. Not-deferrable unique

constraints and primary keys also use a nonunique index if the nonunique index is placed on the key columns before the constraint is enforced.

Constraint States

You can enable or disable integrity constraints at the table level using the CREATE TABLE or ALTER TABLE statement. You can also set constraints to VALIDATE or NOVALIDATE, in any combination with ENABLE or DISABLE, where:

- ENABLE ensures that all incoming data conforms to the constraint
- DISABLE allows incoming data, regardless of whether it conforms to the constraint
- VALIDATE ensures that existing data conforms to the constraint
- NOVALIDATE means that some existing data may not conform to the constraint

In addition:

- ENABLE VALIDATE is the same as ENABLE. The constraint is checked and is guaranteed to hold for all rows.
- ENABLE NOVALIDATE means that the constraint is checked, but it does not have to be true for all rows. This allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid.

  In an ALTER TABLE statement, ENABLE NOVALIDATE resumes constraint checking on disabled constraints without first validating all data in the table.
- DISABLE NOVALIDATE is the same as DISABLE. The constraint is not checked and is not necessarily true.
- DISABLE VALIDATE disables the constraint, drops the index on the constraint, and disallows any modification of the constrained columns.

  For a UNIQUE constraint, the DISABLE VALIDATE state enables you to load data efficiently from a nonpartitioned table into a partitioned table using the EXCHANGE PARTITION clause of the ALTER TABLE statement.

Transitions between these states are governed by the following rules:

- ENABLE implies VALIDATE, unless NOVALIDATE is specified.
- DISABLE implies NOVALIDATE, unless VALIDATE is specified.
- VALIDATE and NOVALIDATE do not have any default implications for the ENABLE and DISABLE states.

- When a unique or primary key moves from the DISABLE state to the ENABLE state, if there is no existing index, a unique index is automatically created. Similarly, when a unique or primary key moves from ENABLE to DISABLE and it is enabled with a unique index, the unique index is dropped.
- When any constraint is moved from the NOVALIDATE state to the VALIDATE state, all data must be checked. (This can be very slow.) However, moving from VALIDATE to NOVALIDATE simply forgets that the data was ever checked.
- Moving a single constraint from the ENABLE NOVALIDATE state to the ENABLE VALIDATE state does not block reads, writes, or other DDL statements. It can be done in parallel.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about how to use the ENABLE, DISABLE, VALIDATE, and NOVALIDATE CONSTRAINT clauses.

Constraint State Modification

You can use the MODIFY CONSTRAINT clause of the ALTER TABLE statement to change the following constraint states:

- DEFERRABLE or NOT DEFERRABLE
- INITIALLY DEFERRED or INITIALLY IMMEDIATE
- RELY or NORELY
- USING INDEX ...
- ENABLE or DISABLE
- VALIDATE or NOVALIDATE
- EXCEPTIONS INTO ...

> **See Also:** *Oracle9i SQL Reference* for information about these constraint states

24

Controlling Database Access

This chapter explains how to control access to an Oracle database. It includes the following sections:

- Introduction to Database Security
- Schemas, Database Users, and Security Domains
- User Authentication
- User Tablespace Settings and Quotas
- The User Group PUBLIC
- User Resource Limits and Profiles
- Overview of Licensing

Introduction to Database Security

Database security entails allowing or disallowing user actions on the database and the objects within it. Oracle uses schemas and security domains to control access to data and to restrict the use of various database resources.

Oracle provides comprehensive discretionary access control. **Discretionary access control** regulates all user access to named objects through privileges. A privilege is permission to access a named object in a prescribed manner; for example, permission to query a table. Privileges are granted to users at the discretion of other users—hence the term **discretionary access control**.

See Also: Chapter 25, "Privileges, Roles, and Security Policies"

Schemas, Database Users, and Security Domains

A **user** (sometimes called a **username**) is a name defined in the database that can connect to and access objects. A **schema** is a named collection of objects, such as tables, views, clusters, procedures, and packages. Schemas and users help database administrators manage database security.

Enterprise users are managed in a directory and can be given access to multiple schemas and databases without having to create an account or schema in each database. This arrangement is simpler for users and for DBAs and also offers better security because their privileges can be altered in one place.

When creating a new database user or altering an existing one, the security administrator must make several decisions concerning a user's security domain. These include:

- Whether user authentication information is maintained by the database, the operating system, or a network authentication service
- Settings for the user's default and temporary tablespaces
- A list of tablespaces accessible to the user, if any, and the associated quotas for each listed tablespace
- The user's resource limit profile; that is, limits on the amount of system resources available to the user
- The privileges, roles, and security policies that provide the user with appropriate access to schema objects needed to perform database operations

This chapter describes the first four security domain options listed.

Note: The information in this chapter applies to all user-defined database users. It does not apply to the special database users SYS and SYSTEM. Settings for these users' security domains should never be altered.

See Also:

- Chapter 25, "Privileges, Roles, and Security Policies"
- *Oracle Advanced Security Administrator's Guide* for more information about enterprise users
- *Oracle9i Database Administrator's Guide* for more information about the special users SYS and SYSTEM, and for information about security administrators

User Authentication

To prevent unauthorized use of a database username, Oracle provides user validation through several different methods for normal database users. You can perform authentication by:

- The operating system
- A network service
- The associated Oracle database
- The Oracle database of a middle-tier application that performs transactions on behalf of the user
- The Secure Socket Layer (SSL) protocol

For simplicity, one method is usually used to authenticate all users of a database. However, Oracle allows use of all methods within the same database instance.

Oracle also encrypts passwords during transmission to ensure the security of network authentication.

Oracle requires special authentication procedures for database administrators, because they perform special database operations.

Authentication by the Operating System

Some operating systems permit Oracle to use information maintained by the operating system to authenticate users. The benefits of authentication by the operating system are:

- Users can connect to Oracle more conveniently, without specifying a username or password. For example, a user can invoke SQL*Plus and skip the username and password prompts by entering

  ```
  SQLPLUS /
  ```

- Control over user authorization is centralized in the operating system. Oracle need not store or manage user passwords. However, Oracle still maintains usernames in the database.
- Username entries in the database and operating system audit trails correspond.

If the operating system is used to authenticate database users, some special considerations arise with respect to distributed database environments and database links.

See Also:

- *Oracle9i Database Administrator's Guide*
- Your Oracle operating system specific documentation for more information about authenticating by way of your operating system

Authentication by the Network

Oracle supports the following methods of authentication by the network.

Third Party-Based Authentication Technologies

If network authentication services are available to you (such as DCE, Kerberos, or SESAME), Oracle can accept authentication from the network service. To use a network authentication service with Oracle, you need Oracle9*i* Enterprise Edition with the Oracle Advanced Security option.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about network authentication. If you use a network authentication service, some special considerations arise for network roles and database links.
- *Oracle Advanced Security Administrator's Guide* for information about the Oracle Advanced Security option

Public Key Infrastructure-Based Authentication

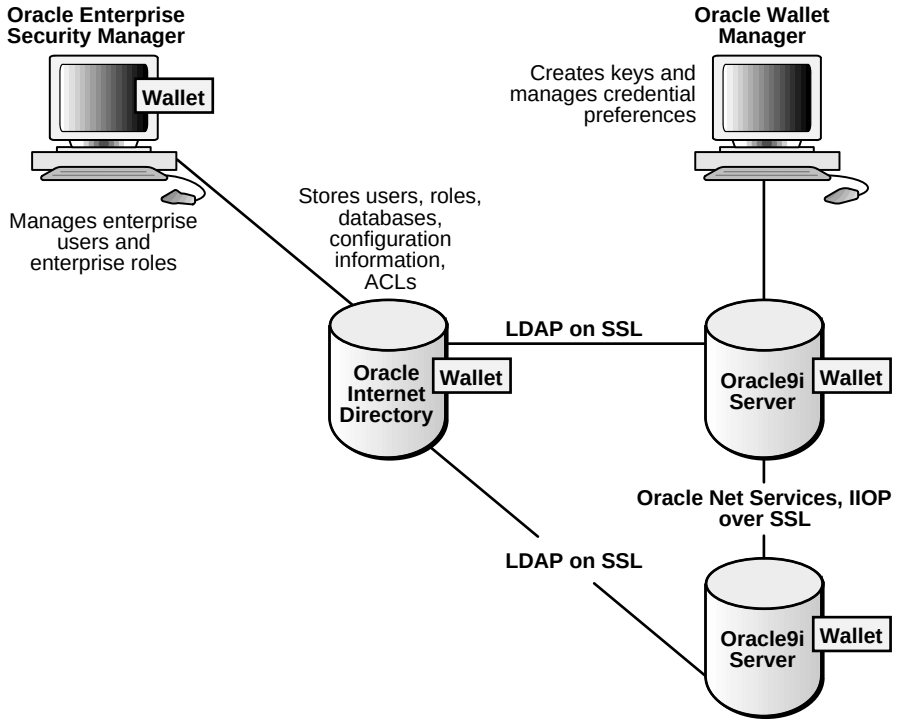
Authentication systems based on public key cryptography systems issue digital certificates to user clients, which use them to authenticate directly to servers in the enterprise without direct involvement of an authentication server. Oracle provides a public key infrastructure (PKI) for using public keys and certificates. It consists of the following components:

- Authentication and secure session key management using Secure Sockets Layer (SSL).
- Oracle Call Interface (OCI) and PL/SQL functions to sign user-specified data using a private key and certificate, and verify the signature on data using a trusted certificate.
- A **trusted certificate**, which is a third-party identity that is trusted. The trust is used when an identity is being validated as the entity it claims to be. Typically, the certificate authorities you trust issue user certificates. If there are several levels of trusted certificates, a trusted certificate at a lower level in the certificate chain does not need to have all its higher level certificates reverified.
- **Oracle wallets**, which are data structures that contain a user private key, a user certificate, and a set of trust points (the list of root certificates the user trusts).
- **Oracle Wallet Manager**, which is a standalone Java application used to manage and edit the security credentials in Oracle wallets. Wallet Manager:
  - Protects user keys
  - Manages X.509v3 certificates on Oracle clients and servers
  - Generates a public-private key pair and creates a certificate request for submission to a certificate authority
  - Installs a certificate for the entity
  - Configures trusted certificates for the entity

  - Opens a wallet to enable access to PKI-based services
  - Creates a wallet that can be opened using the Oracle Enterprise Login Assistant
- X.509v3 certificates that you obtain from a certificate authority outside of Oracle. It is created when an entity's public key is signed by a trusted entity (a certificate authority outside of Oracle). The certificate ensures that the entity's information is correct and the public key belongs to the entity. The certificates are loaded into Oracle wallets to enable authentication.
- **Oracle Enterprise Security Manager**, which provides centralized privilege management to make administration easier and increase your level of security. Oracle Enterprise Security Manager allows you to store and retrieve roles from Oracle Internet Directory if the roles support the Lightweight Directory Access Protocol (LDAP). Oracle Enterprise Security Manager may also allow you to store roles in other LDAP v3-compliant directory servers if they can support the installation of the Oracle schema and related Access Control Lists.
- **Oracle Internet Directory**, which is an LDAP v3-compliant directory built on the Oracle9*i* database. It allows you to manage the user and system configuration environment, including security attributes and privileges, for users authenticated using X.509 certificates. Oracle Internet Directory enforces attribute-level access control, allowing the directory to restrict read, write, or update privileges on specific attributes to specific named users (for example, an enterprise security administrator). It also supports protection and authentication of directory queries and responses through SSL encryption.
- **Oracle Enterprise Login Assistant**, which is a Java-based tool for opening and closing a user wallet in order to enable or disable secure SSL-based communications for an application. This tool provides a subset of the functionality proved by Oracle Wallet Manager. The wallet must be configured with Oracle Wallet Manager first.

Oracle's public key infrastructure is illustrated in Figure 24–1.

Figure 24–1 Oracle Public Key Infrastructure



Note: To use public key infrastructure-based authentication with Oracle, you need Oracle9*i* Enterprise Edition with the Oracle Advanced Security option.

Remote Authentication

Oracle supports remote authentication of users through Remote Dial-In User Service (RADIUS), a standard lightweight protocol used for user authentication, authorization, and accounting.

Note: To use remote authentication of users through RADIUS with Oracle, you need Oracle9*i* Enterprise Edition with the Advanced Security option.

See Also: *Oracle Advanced Security Administrator's Guide* for information about Oracle Advanced Security

Authentication by the Oracle Database

Oracle can authenticate users attempting to connect to a database by using information stored in that database.

When Oracle uses database authentication, you create each user with an associated password. A user provides the correct password when establishing a connection to prevent unauthorized use of the database. Oracle stores a user's password in the data dictionary in an encrypted format. A user can change his or her password at any time.

Password Encryption While Connecting

To protect password confidentiality, Oracle allows you to encrypt passwords during network (client/server and server/server) connections. If you enable this functionality on the client and server machines, Oracle encrypts passwords using a modified DES (Data Encryption Standard) algorithm before sending them across the network. It is strongly recommended that you enable password encryption for connections to protect your passwords from network intrusion.

See Also: *Oracle9i Database Administrator's Guide* for more information about encrypting passwords in network systems

Account Locking

Oracle can lock a user's account if the user fails to login to the system within a specified number of attempts. Depending on how the account is configured, it can be unlocked automatically after a specified time interval or it must be unlocked by the database administrator.

The `CREATE PROFILE` statement configures the number of failed logins a user can attempt and the amount of time the account remains locked before automatic unlock.

The database administrator can also lock accounts manually. When this occurs, the account cannot be unlocked automatically but must be unlocked explicitly by the database administrator.

See Also: "Profiles" on page 24-21

Password Lifetime and Expiration

Password lifetime and expiration options allow the database administrator to specify a lifetime for passwords, after which time they expire and must be changed before a login to the account can be completed. On first attempt to login to the database account after the password expires, the user's account enters the grace period, and a warning message is issued to the user every time the user tries to login until the grace period is over.

The user is expected to change the password within the grace period. If the password is not changed within the grace period, the account is locked and no further logins to that account are allowed without assistance by the database administrator.

The database administrator can also set the password state to expired. When this happens, the user's account status is changed to expired, and the user or the database administrator must change the password before the user can log into the database.

Password History

The password history option checks each newly specified password to ensure that a password is not reused for the specified amount of time or for the specified number of password changes. The database administrator can configure the rules for password reuse with `CREATE PROFILE` statements.

Password Complexity Verification

Complexity verification checks that each password is complex enough to provide reasonable protection against intruders who try to break into the system by guessing passwords.

The Oracle default password complexity verification routine requires that each password:

- Be a minimum of four characters in length
- Not equal the userid

- Include at least one alphabet character, one numeric character, and one punctuation mark
- Not match any word on an internal list of simple words like welcome, account, database, user, and so on
- Differ from the previous password by at least three characters

Multitier Authentication and Authorization

In a multitier environment, Oracle controls the security of middle-tier applications by limiting their privileges, preserving client identities through all tiers, and auditing actions taken on behalf of clients. In applications that use a heavy middle tier, such as a transaction processing monitor, it is important to be able to preserve the identity of the client connecting to the middle tier. Yet one advantage of a middle tier is **connection pooling**, which allows multiple users to access a data server without each of them needing a separate connection. In such environments, you need to be able to set up and break down connections very quickly. For these environments, Oracle offers the creation of **lightweight sessions** through the Oracle Call Interface. These lightweight sessions allow each user to be authenticated by a database password without the overhead of a separate database connection, as well as preserving the identity of the real user through the middle tier.

You can create lightweight sessions with or without passwords. If a middle tier is outside or on a firewall, it would be appropriate to establish the lightweight session with passwords for each lightweight user session. For an internal application server, it might be appropriate to create a lightweight session that does not require passwords.

Clients, Application Servers, and Database Servers

In a multitier architecture environment, an application server provides data for clients and serves as an interface between clients and one or more database servers.

This architecture allows you to use an application server to validate the credentials of a client, such as a web browser. In addition, the database server can audit operations performed by the application server and operations performed by the application server on behalf of the client. For example, an operation performed by the application server on behalf of the client might be a request for information to be displayed on the client whereas an operation performed by the application server might be a request for a connection to the database server.

Authentication in a multitier environment is based on trust regions, including the following:

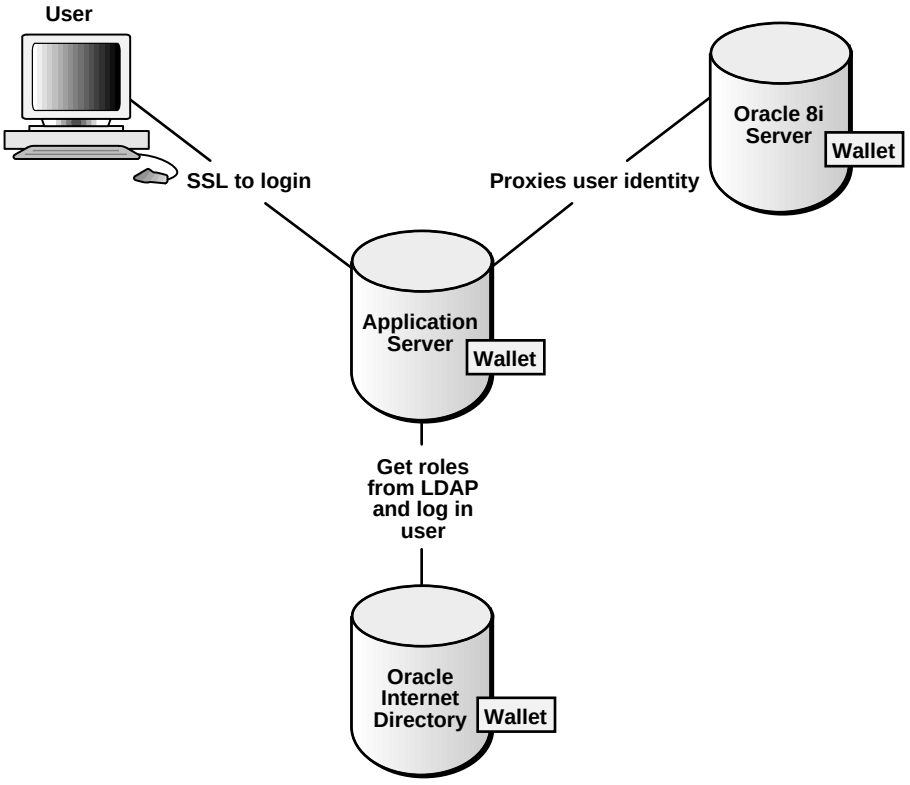
- The client provides proof of authentication to the application server, typically using a password or an X.509 certificate.
- The application server verifies the client authentication and then authenticates itself to the database server.
- The database server checks the application server authentication, verifies that the client exists, and verifies that the application server has the privilege to connect for this client.

Application servers can also enable roles for the client on whose behalf it is connecting. The application server can obtain these roles from a directory, which thus serves as an authorization repository. The application server can only request that these roles be enabled. The database verifies that:

- The client has these roles by checking its internal role repository.
- The application server has the privilege to connect on behalf of the user, using these roles for the user.

Figure 24–2 shows an example of multitier authentication.

Figure 24–2 Multitier Authentication



Security Issues for Middle-Tier Applications

There are a number of security issues for middle-tier applications:

Accountability	The database server must be able to distinguish between the actions of a client and the actions an application takes on behalf of a client. It must be possible to audit both kinds of actions.
Differentiation	The database server must be able to distinguish between a web server transaction, a web server transaction on behalf of a browser client, and a client accessing the database directly.
Least privilege	Users and middle tiers should be given the fewest privileges necessary to do their jobs.

Identity Issues in a Multitier Environment

Multitier authentication maintains the identify of the client through all tiers of the connection. This is necessary because if the identity of the originating client is lost, it is not possible to maintain useful audit records. In addition, it is not possible to distinguish operations performed by the application server on behalf of the client from those done by the application server for itself.

Restricted Privileges in a Multitier Environment

Privileges in a multitier environment are limited to what is necessary to perform the requested operation.

Client Privileges Client privileges are as limited as possible in a multitier environment. Operations are performed on behalf of the client by the application server.

Application Server Privileges Application server privileges in a multitier environment are limited so that the application server cannot perform unwanted or unneeded operations while performing a client operation.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about multitier authentication

Authentication by the Secure Socket Layer Protocol

The Secure Socket Layer (SSL) protocol is an application layer protocol. It can be used for user authentication to a database, independent of global user management

in Oracle Internet Directory. That is, users can use SSL to authenticate to the database without implying anything about their directory access. However, if you wish to use the enterprise user functionality to manage users and their privileges in a directory, the user must use SSL to authenticate to the database. A parameter in the initialization file governs which use of SSL is expected.

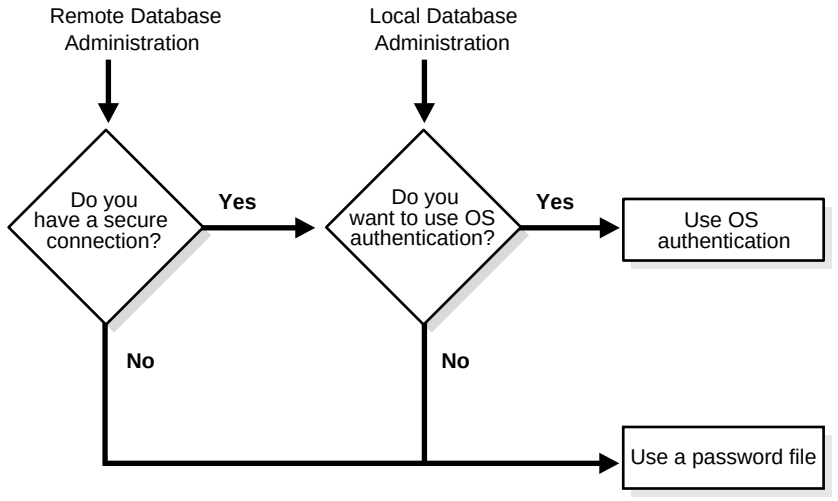
Authentication of Database Administrators

Database administrators perform special operations (such as shutting down or starting up a database) that should not be performed by normal database users. Oracle provides a more secure authentication scheme for database administrator usernames.

You can choose between operating system authentication or password files to authenticate database administrators.

Figure 24–3 illustrates the choices you have for database administrator authentication schemes, depending on whether you administer your database locally (on the same machine on which the database resides) or if you administer many different database machines from a single remote client.

Figure 24–3 Database Administrator Authentication Methods



On most operating systems, operating system authentication for database administrators involves placing the operating system username of the database administrator in a special group (on UNIX systems, this is the **dba** group) or giving that operating system username a special process right.

The database uses password files to keep track of database usernames who have been granted the SYSDBA and SYSOPER privileges. These privileges allow database administrators to perform the following actions.

SYSOPER	Permits you to perform STARTUP, SHUTDOWN, ALTER DATABASE OPEN/MOUNT, ALTER DATABASE BACKUP, ARCHIVE LOG, and RECOVER, and includes the RESTRICTED SESSION privilege.
SYSDBA	Contains all system privileges with ADMIN OPTION, and the SYSOPER system privilege. Permits CREATE DATABASE and time-based recovery.

See Also:

- Your Oracle operating system specific documentation for information about operating system authentication of database administrators
- *Oracle9i Database Administrator's Guide*

User Tablespace Settings and Quotas

As part of every user's security domain, the database administrator can set several options regarding tablespace use:

- Default Tablespace Option
- Temporary Tablespace Option
- Tablespace Access and Quotas

Default Tablespace Option

When a user creates a schema object without specifying a tablespace to contain the object, Oracle places the object in the user's default tablespace. You set a user's default tablespace when the user is created, and you can change it after the user has been created.

Temporary Tablespace Option

When a user executes a SQL statement that requires the creation of a temporary segment, Oracle allocates that segment in the user's temporary tablespace.

Tablespace Access and Quotas

You can assign to each user a **tablespace quota** for any tablespace of the database. Doing so can accomplish two things:

- You allow the user to use the specified tablespace to create schema objects, if the user has the appropriate privileges.
- You can limit the amount of space allocated for storage of a user's schema objects in the specified tablespace.

By default, each user has no quota on any tablespace in the database. Therefore, if the user has the privilege to create some type of schema object, he or she must also have been either assigned a tablespace quota in which to create the object or been

given the privilege to create that object in the schema of another user who was assigned a sufficient tablespace quota.

You can assign two types of tablespace quotas to a user: a quota for a specific amount of disk space in the tablespace (specified in bytes, kilobytes, or megabytes), or a quota for an unlimited amount of disk space in the tablespace. You should assign specific quotas to prevent a user's objects from consuming too much space in a tablespace.

Tablespace quotas and temporary segments have no effect on each other:

- Temporary segments do not consume any quota that a user might possess. The schema objects that Oracle automatically creates in temporary segments are owned by `SYS` and therefore are not subject to quotas.
- Temporary segments can be created in a tablespace for which a user has no quota.

You can assign a tablespace quota to a user when you create that user, and you can change that quota or add a different quota later.

Revoke a user's tablespace access by altering the user's current quota to zero. With a quota of zero, the user's objects in the revoked tablespace remain, but the objects cannot be allocated any new space.

The User Group PUBLIC

Each database contains a user group called `PUBLIC`. The `PUBLIC` user group provides public access to specific schema objects, such as tables and views, and provides all users with specific system privileges. Every user automatically belongs to the `PUBLIC` user group.

As members of `PUBLIC`, users can see (select from) all data dictionary tables prefixed with `USER` and `ALL`. Additionally, a user can grant a privilege or a role to `PUBLIC`. All users can use the privileges granted to `PUBLIC`.

You can grant or revoke any system privilege, object privilege, or role to `PUBLIC`. However, to maintain tight security over access rights, grant only privileges and roles that are of interest to all users to `PUBLIC`.

Granting and revoking some system and object privileges to and from `PUBLIC` can cause every view, procedure, function, package, and trigger in the database to be recompiled.

`PUBLIC` has the following restrictions:

- You cannot assign tablespace quotas to PUBLIC, although you can assign the UNLIMITED TABLESPACE system privilege to PUBLIC.
- You can create database links and synonyms as PUBLIC (using CREATE PUBLIC DATABASE LINK/SYNONYM), but no other schema object can be owned by PUBLIC. For example, the following statement is not legal:

```
CREATE TABLE public.emp ... ;
```

Note: Rollback segments can be created with the keyword PUBLIC, but these are not owned by the PUBLIC user group. All rollback segments are owned by SYS.

See Also:

- Chapter 3, "Data Blocks, Extents, and Segments"
- Chapter 25, "Privileges, Roles, and Security Policies"

User Resource Limits and Profiles

You can set limits on the amount of various system resources available to each user as part of a user's security domain. By doing so, you can prevent the uncontrolled consumption of valuable system resources such as CPU time.

This resource limit feature is very useful in large, multiuser systems, where system resources are very expensive. Excessive consumption of these resources by one or more users can detrimentally affect the other users of the database. In single-user or small-scale multiuser database systems, the system resource feature is not as important, because users' consumption of system resources is less likely to have detrimental impact.

You manage a user's resource limits and password management preferences with his or her profile—a named set of resource limits that you can assign to that user. Each Oracle database can have an unlimited number of profiles. Oracle allows the security administrator to enable or disable the enforcement of profile resource limits universally.

If you set resource limits, a slight degradation in performance occurs when users create sessions. This is because Oracle loads all resource limit data for the user when a user connects to a database.

See Also: *Oracle9i Database Administrator's Guide* for information about security administrators

Types of System Resources and Limits

Oracle can limit the use of several types of system resources, including CPU time and logical reads. In general, you can control each of these resources at the session level, the call level, or both:

Session level

Each time a user connects to a database, a session is created. Each session consumes CPU time and memory on the computer that executes Oracle. You can set several resource limits at the session level.

If a user exceeds a session-level resource limit, Oracle terminates (rolls back) the current statement and returns a message indicating the session limit has been reached. At this point, all previous statements in the current transaction are intact, and the only operations the user can perform are `COMMIT, ROLLBACK,` or disconnect (in this case, the current transaction is committed). All other operations produce an error. Even after the transaction is committed or rolled back, the user can accomplish no more work during the current session.

Call level

Each time a SQL statement is executed, several steps are taken to process the statement. During this processing, several calls are made to the database as part of the different execution phases. To prevent any one call from using the system excessively, Oracle allows you to set several resource limits at the call level.

If a user exceeds a call-level resource limit, Oracle halts the processing of the statement, rolls back the statement, and returns an error. However, all previous statements of the current transaction remain intact, and the user's session remains connected.

CPU Time

When SQL statements and other types of calls are made to Oracle, an amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data or a

runaway query can potentially consume a large amount of CPU time, reducing CPU time available for other processing.

To prevent uncontrolled use of CPU time, you can limit the CPU time for each call and the total amount of CPU time used for Oracle calls during a session. The limits are set and measured in CPU one-hundredth seconds (0.01 seconds) used by a call or a session.

Logical Reads

Input/output (I/O) is one of the most expensive operations in a database system. SQL statements that are I/O intensive can monopolize memory and disk use and cause other database operations to compete for these resources.

To prevent single sources of excessive I/O, Oracle let you limit the logical data block reads for each call and for each session. Logical data block reads include data block reads from both memory and disk. The limits are set and measured in number of block reads performed by a call or during a session.

Other Resources

Oracle also provides for the limitation of several other resources at the session level:

- You can limit the number of **concurrent sessions for each user**. Each user can create only up to a predefined number of concurrent sessions.
- You can limit the **idle time** for a session. If the time between Oracle calls for a session reaches the idle time limit, the current transaction is rolled back, the session is aborted, and the resources of the session are returned to the system. The next call receives an error that indicates the user is no longer connected to the instance. This limit is set as a number of elapsed minutes.

  Note: Shortly after a session is aborted because it has exceeded an idle time limit, the process monitor (PMON) background process cleans up after the aborted session. Until PMON completes this process, the aborted session is still counted in any session/user resource limit.

- You can limit the elapsed connect time for each session. If a session's duration exceeds the elapsed time limit, the current transaction is rolled back, the session is dropped, and the resources of the session are returned to the system. This limit is set as a number of elapsed minutes.

> **Note:** Oracle does not constantly monitor the elapsed idle time or elapsed connection time. Doing so would reduce system performance. Instead, it checks every few minutes. Therefore, a session can exceed this limit slightly (for example, by five minutes) before Oracle enforces the limit and aborts the session.

- You can limit the amount of private SGA space (used for private SQL areas) for a session. This limit is only important in systems that use the shared server configuration. Otherwise, private SQL areas are located in the PGA. This limit is set as a number of bytes of memory in an instance's SGA. Use the characters **K** or **M** to specify kilobytes or megabytes.

  See Also: *Oracle9i Database Administrator's Guide* for instructions about enabling and disabling resource limits

Profiles

A profile is a named set of specified resource limits that can be assigned to a valid username of an Oracle database. Profiles provide for easy management of resource limits. Profiles are also the way in which you administer password policy.

When to Use Profiles

You need to create and manage user profiles only if resource limits are a requirement of your database security policy. To use profiles, first categorize the related types of users in a database. Just as roles are used to manage the privileges of related users, profiles are used to manage the resource limits of related users. Determine how many profiles are needed to encompass all types of users in a database and then determine appropriate resource limits for each profile.

Determine Values for Resource Limits of a Profile

Before creating profiles and setting the resource limits associated with them, you should determine appropriate values for each resource limit. You can base these values on the type of operations a typical user performs. For example, if one class of user does not normally perform a high number of logical data block reads, then set the `LOGICAL_READS_PER_SESSION` and `LOGICAL_READS_PER_CALL` limits conservatively.

Usually, the best way to determine the appropriate resource limit values for a given user profile is to gather historical information about each type of resource usage.

For example, the database or security administrator can use the `AUDIT SESSION` clause to gather information about the limits `CONNECT_TIME`, `LOGICAL_READS_PER_SESSION`, and `LOGICAL_READS_PER_CALL`.

You can gather statistics for other limits using the Monitor feature of Oracle Enterprise Manager (or SQL*Plus), specifically the Statistics monitor.

> **See Also:** Chapter 26, "Auditing"

Overview of Licensing

Oracle is usually licensed for use by a maximum number of named users or by a maximum number of concurrently connected users. The database administrator (DBA) is responsible for ensuring that the site complies with its license agreement. Oracle's licensing facility helps the DBA monitor system use by tracking and limiting the number of sessions concurrently connected to an instance or the number of users created in a database.

If the DBA discovers that more than the licensed number of sessions need to connect, or more than the licensed number of users need to be created, he or she can upgrade the Oracle license to raise the appropriate limit.

> **Note:** When Oracle is embedded in an Oracle application, run on some older operating systems, or purchased for use in some countries, it is not licensed for either a set number of sessions or a set group of users. In such cases only, the Oracle licensing mechanisms do not apply and should remain disabled.

The following sections explain the two major types of licensing available for Oracle.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about licensing

Concurrent Usage Licensing

In **concurrent usage licensing**, the license specifies a number of **concurrent users**, which are sessions that can be connected concurrently to the database on the specified computer at any time. This number includes all batch processes and online users. If a single user has multiple concurrent sessions, each session counts separately in the total number of sessions. If multiplexing software (such as a TP monitor) is used to reduce the number of sessions directly connected to the

database, the number of concurrent users is the number of distinct inputs to the multiplexing front end.

The concurrent usage licensing mechanism allows a DBA to:

- Set a limit on the number of concurrent sessions that can connect to an instance by setting the `LICENSE_MAX_SESSIONS` parameter. Once this limit is reached, only users who have the `RESTRICTED SESSION` system privilege can connect to the instance. This allows DBA to kill unneeded sessions, allowing other sessions to connect.
- Set a warning limit on the number of concurrent sessions that can connect to an instance by setting the `LICENSE_SESSIONS_WARNING` parameter. After the warning limit is reached, Oracle allows additional sessions to connect (up to the maximum limit described previously). However, Oracle sends a warning message to any user who connects with `RESTRICTED SESSION` privilege and records a warning message in the database's `ALERT` file.

The DBA can set these limits in the database's parameter file so that they take effect when the instance starts and can change them while the instance is running (using the `ALTER SYSTEM` statement). The latter is useful for databases that cannot be taken offline.

The session licensing mechanism allows a DBA to check the current number of connected sessions and the maximum number of concurrent sessions since the instance started. The `V$LICENSE` view shows the current settings for the license limits, the current number of sessions, and the highest number of concurrent sessions since the instance started (the session high water mark). The DBA can use this information to evaluate the system's licensing needs and plan for system upgrades.

For instances running with Oracle9*i* Real Application Clusters, each instance can have its own concurrent usage limit and warning limit. The sum of the instances' limits must not exceed the site's concurrent usage license.

The concurrent usage limits apply to all user sessions, including sessions created for incoming database links. They do not apply to sessions created by Oracle or to recursive sessions. Sessions that connect through external multiplexing software are not counted separately by the Oracle licensing mechanism, although each contributes individually to the Oracle license total. The DBA is responsible for taking these sessions into account.

Named User Licensing

In **named user licensing**, the license specifies a number of named users, where a **named user** is an individual who is authorized to use Oracle on the specified computer. No limit is set on the number of sessions each user can have concurrently, or on the number of concurrent sessions for the database.

Named user licensing allows a DBA to set a limit on the number of users that are defined in a database, including users connected through database links. After this limit is reached, no one can create a new user. This mechanism assumes that each person accessing the database has a unique user name in the database and that no two (or more) people share a user name.

The DBA can set this limit in the database's parameter file so that it takes effect when the instance starts and can change it while the instance is running (using the `ALTER SYSTEM` statement). The latter is useful for databases that cannot be taken offline.

If multiple instances connect to the same database in Oracle9*i* Real Application Clusters, all instances connected to the same database should have the same named user limit.

See Also:

- *Oracle9i Real Application Clusters Concepts*
- *Oracle9i Real Application Clusters Installation and Configuration*

25

Privileges, Roles, and Security Policies

This chapter explains how you can control users' ability to execute system operations and to access schema objects by using privileges, roles, and security policies. The chapter includes:

- Introduction to Privileges
- Introduction to Roles
- Fine-Grained Access Control
- Application Context
- Secure Application Roles

Introduction to Privileges

A **privilege** is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to:

- Connect to the database (create a session)
- Create a table
- Select rows from another user's table
- Execute another user's stored procedure

You grant privileges to users so these users can accomplish tasks required for their job. You should grant a privilege only to a user who absolutely requires the privilege to accomplish necessary work. Excessive granting of unnecessary privileges can compromise security. A user can receive a privilege in two different ways:

- You can grant privileges to users explicitly. For example, you can explicitly grant the privilege to insert records into the `EMP` table to the user `SCOTT`.
- You can also grant privileges to a role (a named group of privileges), and then grant the role to one or more users. For example, you can grant the privileges to select, insert, update, and delete records from the `EMP` table to the role named `CLERK`, which in turn you can grant to the users `SCOTT` and `BRIAN`.

Because roles allow for easier and better management of privileges, you should normally grant privileges to roles and not to specific users.

There are two distinct categories of privileges:

- System privileges
- Schema object privileges

> **See Also:** *Oracle9i Database Administrator's Guide* for a complete list of all system and schema object privileges, as well as instructions for privilege management

System Privileges

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges. There are over 60 distinct system privileges.

Grant and Revoke System Privileges

You can grant or revoke system privileges to users and roles. If you grant system privileges to roles, then you can use the roles to manage system privileges. For example, roles permit privileges to be made selectively available.

> **Note:** In general, you grant system privileges only to administrative personnel and application developers. End users normally do not require the associated capabilities.

Use either of the following to grant or revoke system privileges to users and roles:

- The Grant System Privileges / Roles dialog box and Revoke System Privileges / Roles dialog box of Oracle Enterprise Manager
- The SQL statements `GRANT` and `REVOKE`

Who Can Grant or Revoke System Privileges?

Only users who have been granted a specific system privilege with the `ADMIN OPTION` or users with the `GRANT ANY PRIVILEGE` system privilege can grant or revoke system privileges to other users.

Schema Object Privileges

A **schema object privilege** is a privilege or right to perform a particular action on a specific schema object:

- Table
- View
- Sequence
- Procedure
- Function
- Package

Different object privileges are available for different types of schema objects. For example, the privilege to delete rows from the `DEPT` table is an object privilege.

Some schema objects, such as clusters, indexes, triggers, and database links, do not have associated object privileges. Their use is controlled with system privileges. For

example, to alter a cluster, a user must own the cluster or have the ALTER ANY CLUSTER system privilege.

A schema object and its synonym are equivalent with respect to privileges. That is, the object privileges granted for a table, view, sequence, procedure, function, or package apply whether referencing the base object by name or using a synonym.

For example, assume there is a table JWARD.EMP with a synonym named JWARD.EMPLOYEE and the user JWARD issues the following statement:

```
GRANT SELECT ON emp TO swilliams;
```

The user SWILLIAMS can query JWARD.EMP by referencing the table by name or using the synonym JWARD.EMPLOYEE:

```
SELECT * FROM jward.emp;
SELECT * FROM jward.employee;
```

If you grant object privileges on a table, view, sequence, procedure, function, or package to a **synonym** for the object, the effect is the same as if no synonym were used. For example, if JWARD wanted to grant the SELECT privilege for the EMP table to SWILLIAMS, JWARD could issue either of the following statements:

```
GRANT SELECT ON emp TO swilliams;
GRANT SELECT ON employee TO swilliams;
```

If a synonym is dropped, all grants for the underlying schema object remain in effect, even if the privileges were granted by specifying the dropped synonym.

Grant and Revoke Schema Object Privileges

Schema object privileges can be granted to and revoked from users and roles. If you grant object privileges to roles, you can make the privileges selectively available. Object privileges for users and roles can be granted or revoked using the following:

- The SQL statements GRANT and REVOKE, respectively
- The Add Privilege to Role/User dialog box and the Revoke Privilege from Role/User dialog box of Oracle Enterprise Manager.

Who Can Grant Schema Object Privileges?

A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. If the grant includes the GRANT OPTION of the

GRANT statement, the grantee can further grant the object privilege to other users. Otherwise, the grantee can use the privilege but cannot grant it to other users.

> **See Also:** *Oracle9i SQL Reference*

Table Security

Schema object privileges for tables allow table security at the level of DML and DDL operations.

Data Manipulation Language Operations

You can grant privileges to use the DELETE, INSERT, SELECT, and UPDATE DML operations on a table or view. Grant these privileges only to users and roles that need to query or manipulate a table's data.

You can restrict INSERT and UPDATE privileges for a table to specific columns of the table. With selective INSERT, a privileged user can insert a row with values for the selected columns. All other columns receive NULL or the column's default value. With selective UPDATE, a user can update only specific column values of a row. Selective INSERT and UPDATE privileges are used to restrict a user's access to sensitive data.

For example, if you do not want data entry users to alter the SAL column of the employee table, selective INSERT and/or UPDATE privileges can be granted that exclude the SAL column. Alternatively, a view that excludes the SAL column could satisfy this need for additional security.

> **See Also:** *Oracle9i SQL Reference* for more information about these DML operations

Data Definition Language Operations

The ALTER, INDEX, and REFERENCES privileges allow DDL operations to be performed on a table. Because these privileges allow other users to alter or create dependencies on a table, you should grant privileges conservatively. A user attempting to perform a DDL operation on a table may need additional system or object privileges. For example, to create a trigger on a table, the user requires both the ALTER TABLE object privilege for the table and the CREATE TRIGGER system privilege.

As with the INSERT and UPDATE privileges, the REFERENCES privilege can be granted on specific columns of a table. The REFERENCES privilege enables the grantee to use the table on which the grant is made as a parent key to any foreign

keys that the grantee wishes to create in his or her own tables. This action is controlled with a special privilege because the presence of foreign keys restricts the data manipulation and table alterations that can be done to the parent key.
A column-specific `REFERENCES` privilege restricts the grantee to using the named columns (which, of course, must include at least one primary or unique key of the parent table).

> **See Also:** Chapter 23, "Data Integrity" for more information about primary keys, unique keys, and integrity constraints

View Security

Schema object privileges for views allow various DML operations, which actually affect the base tables from which the view is derived. DML object privileges for tables can be applied similarly to views.

Privileges Required to Create Views

To create a view, you must meet the following requirements:

- You must have been granted one of the following system privileges, either explicitly or through a role:
  - The `CREATE VIEW` system privilege (to create a view in your schema)
  - The `CREATE ANY VIEW` system privilege (to create a view in another user's schema)
- You must have been explicitly granted one of the following privileges:
  - The `SELECT`, `INSERT`, `UPDATE`, or `DELETE` object privileges on all base objects underlying the view
  - The `SELECT ANY TABLE`, `INSERT ANY TABLE`, `UPDATE ANY TABLE`, or `DELETE ANY TABLE` system privileges
- Additionally, in order to grant other users access to your view, you must have received object privileges to the base objects with the `GRANT OPTION` clause or appropriate system privileges with the `ADMIN OPTION` clause. If you have not, grantees cannot access your view.

> **See Also:** *Oracle9i SQL Reference*

Increase Table Security with Views

To use a view, you require appropriate privileges only for the view itself. You do not require privileges on base objects underlying the view.

Views add two more levels of security for tables, column-level security and value-based security:

- A view can provide access to selected columns of base tables. For example, you can define a view on the EMP table to show only the EMPNO, ENAME, and MGR columns:

```
CREATE VIEW emp_mgr AS
    SELECT ename, empno, mgr FROM emp;
```

- A view can provide value-based security for the information in a table. A WHERE clause in the definition of a view displays only selected rows of base tables. Consider the following two examples:

```
CREATE VIEW lowsal AS
    SELECT * FROM emp
    WHERE sal < 10000;
```

  The LOWSAL view allows access to all rows of the EMP table that have a salary value less than 10000. Notice that all columns of the EMP table are accessible in the LOWSAL view.

```
CREATE VIEW own_salary AS
    SELECT ename, sal
    FROM emp
    WHERE ename = USER;
```

  In the OWN_SALARY view, only the rows with an ENAME that matches the current user of the view are accessible. The OWN_SALARY view uses the USER pseudocolumn, whose values always refer to the current user. This view combines both column-level security and value-based security.

Procedure Security

The only **schema object privilege** for procedures, including standalone procedures and functions as well as packages, is EXECUTE. Grant this privilege only to users who need to execute a procedure or compile another procedure that calls it.

Procedure Execution and Security Domains

A user with the EXECUTE object privilege for a specific procedure can execute the procedure or compile a program unit that references the procedure. No runtime privilege check is made when the procedure is called. A user with the EXECUTE ANY PROCEDURE system privilege can execute any procedure in the database.

A user can be granted privileges through roles to execute procedures.

Additional privileges on referenced objects are required for invoker-rights procedures, but not for definer-rights procedures.

See Also: "PL/SQL Blocks and Roles" on page 25-21

Definer Rights A user of a definer-rights procedure requires only the privilege to execute the procedure and no privileges on the underlying objects that the procedure accesses, because a definer-rights procedure operates under the security domain of the user who owns the procedure, regardless of who is executing it. The procedure's owner must have all the necessary object privileges for referenced objects. Fewer privileges have to be granted to users of a definer-rights procedure, resulting in tighter control of database access.

You can use definer-rights procedures to control access to private database objects and add a level of database security. By writing a definer-rights procedure and granting only EXECUTE privilege to a user, the user can be forced to access the referenced objects only through the procedure.

At runtime, the privileges of the owner of a definer-rights stored procedure are always checked before the procedure is executed. If a necessary privilege on a referenced object has been revoked from the owner of a definer-rights procedure, then the procedure cannot be executed by the owner or any other user.

Note: Trigger execution follows the same patterns as definer-rights procedures. The user executes a SQL statement, which that user is privileged to execute. As a result of the SQL statement, a trigger is fired. The statements within the triggered action temporarily execute under the security domain of the user that owns the trigger.

See Also: Chapter 18, "Triggers"

Invoker Rights An invoker-rights procedure executes with all of the invoker's privileges. Roles are enabled unless the invoker-rights procedure was called directly or indirectly by a definer-rights procedure. A user of an invoker-rights procedure needs privileges (either directly or through a role) on objects that the procedure accesses through external references that are resolved in the invoker's schema.

The invoker needs privileges at runtime to access program references embedded in DML statements or dynamic SQL statements, because they are effectively recompiled at runtime.

For all other external references, such as direct PL/SQL function calls, the owner's privileges are checked at compile time, and no runtime check is made. Therefore, the user of an invoker-rights procedure needs no privileges on external references outside DML or dynamic SQL statements. Alternatively, the developer of an invoker-rights procedure only needs to grant privileges on the procedure itself, not on all objects directly referenced by the invoker-rights procedure.

Many packages provided by Oracle, such as most of the `DBMS_*` packages, run with invoker rights—they do not run as the owner `(SYS)` but rather as the current user. However, some exceptions exist such as the `DBMS_RLS` package.

You can create a software bundle that consists of multiple program units, some with definer rights and others with invoker rights, and restrict the program entry points *(***controlled step-in***)*. A user who has the privilege to execute an entry-point procedure can also execute internal program units indirectly, but cannot directly call the internal programs.

See Also:

- "Fine-Grained Access Control" on page 25-24
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for detailed documentation of the Oracle supplied packages

System Privileges Needed to Create or Alter a Procedure

To create a procedure, a user must have the `CREATE PROCEDURE` or `CREATE ANY PROCEDURE` **system privilege**. To alter a procedure, that is, to manually recompile a procedure, a user must own the procedure or have the `ALTER ANY PROCEDURE` system privilege.

The user who owns the procedure also must have privileges for schema objects referenced in the procedure body. To create a procedure, you must have been explicitly granted the necessary privileges (system or object) on all objects referenced by the procedure. You cannot have obtained the required privileges

through roles. This includes the EXECUTE privilege for any procedures that are called inside the procedure being created.

Triggers also require that privileges to referenced objects be granted explicitly to the trigger owner. Anonymous PL/SQL blocks can use any privilege, whether the privilege is granted explicitly or through a role.

Packages and Package Objects

A user with the EXECUTE object privilege for a package can execute any public procedure or function in the package and access or modify the value of any public package variable. Specific EXECUTE privileges cannot be granted for a package's constructs. Therefore, you may find it useful to consider two alternatives for establishing security when developing procedures, functions, and packages for a database application. These alternatives are described in the following examples.

Packages and Package Objects Example 1 This example shows four procedures created in the bodies of two packages.

```
CREATE PACKAGE BODY hire_fire AS
  PROCEDURE hire(...) IS
    BEGIN
      INSERT INTO emp . . .
    END hire;
  PROCEDURE fire(...) IS
    BEGIN
      DELETE FROM emp . . .
    END fire;
END hire_fire;

CREATE PACKAGE BODY raise_bonus AS
  PROCEDURE give_raise(...) IS
    BEGIN
      UPDATE EMP SET sal = . . .
    END give_raise;
  PROCEDURE give_bonus(...) IS
    BEGIN
      UPDATE EMP SET bonus = . . .
    END give_bonus;
END raise_bonus;
```

Access to execute the procedures is given by granting the EXECUTE privilege for the package, using the following statements:

```
GRANT EXECUTE ON hire_fire TO big_bosses;
```

```
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Granting EXECUTE privilege granted for a package provides uniform access to all package objects.

Packages and Package Objects Example 2 This example shows four procedure definitions within the body of a single package. Two additional standalone procedures and a package are created specifically to provide access to the procedures defined in the main package.

```
CREATE PACKAGE BODY employee_changes AS
  PROCEDURE change_salary(...) IS BEGIN ... END;
  PROCEDURE change_bonus(...) IS BEGIN ... END;
  PROCEDURE insert_employee(...) IS BEGIN ... END;
  PROCEDURE delete_employee(...) IS BEGIN ... END;
END employee_changes;

CREATE PROCEDURE hire
  BEGIN
    employee_changes.insert_employee(...)
  END hire;

CREATE PROCEDURE fire
  BEGIN
    employee_changes.delete_employee(...)
  END fire;

PACKAGE raise_bonus IS
  PROCEDURE give_raise(...) AS
    BEGIN
      employee_changes.change_salary(...)
    END give_raise;

  PROCEDURE give_bonus(...)
    BEGIN
      employee_changes.change_bonus(...)
    END give_bonus;
```

Using this method, the procedures that actually do the work (the procedures in the EMPLOYEE_CHANGES package) are defined in a single package and can share declared global variables, cursors, on so on. By declaring top-level procedures HIRE and FIRE, and an additional package RAISE_BONUS, you can grant selective EXECUTE privileges on procedures in the main package:

```
GRANT EXECUTE ON hire, fire TO big_bosses;
```

```
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Type Security

This section describes privileges for types, methods, and objects.

System Privileges for Named Types

Oracle9*i* defines system privileges shown in Table 25–1 for named types (object types, VARRAYs, and nested tables):

Table 25–1 System Privileges for Named Types

Privilege	Allows you to...
CREATE TYPE	Create named types in your own schemas.
CREATE ANY TYPE	Create a named type in any schema.
ALTER ANY TYPE	Alter a named type in any schema.
DROP ANY TYPE	Drop a named type in any schema.
EXECUTE ANY TYPE	Use and reference a named type in any schema.

The CONNECT and RESOURCE roles include the CREATE TYPE system privilege. The DBA role includes all of these privileges.

Object Privileges

The only object privilege that applies to named types is EXECUTE. If the EXECUTE privilege exists on a named type, a user can use the named type to:

- Define a table
- Define a column in a relational table
- Declare a variable or parameter of the named type

The EXECUTE privilege permits a user to invoke the type's methods, including the type constructor. This is similar to EXECUTE privilege on a stored PL/SQL procedure.

Method Execution Model

Method execution is the same as any other stored PL/SQL procedure.

See Also: "Procedure Security" on page 25-7

Privileges Required to Create Types and Tables Using Types

To create a type, you must meet the following requirements:

- You must have the CREATE TYPE system privilege to create a type in your schema or the CREATE ANY TYPE system privilege to create a type in another user's schema. These privileges can be acquired explicitly or through a role.
- The owner of the type must have been explicitly granted the EXECUTE object privileges to access all other types referenced within the definition of the type, or have been granted the EXECUTE ANY TYPE system privilege. The owner cannot have obtained the required privileges through roles.
- If the type owner intends to grant access to the type to other users, the owner must have received the EXECUTE privileges to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, the type owner has insufficient privileges to grant access on the type to other users.

To create a table using types, you must meet the requirements for creating a table and these additional requirements:

- The owner of the table must have been explicitly granted the EXECUTE object privileges to access all types referenced by the table, or have been granted the EXECUTE ANY TYPE system privilege. The owner cannot have obtained the required privileges through roles.
- If the table owner intends to grant access to the table to other users, the owner must have received the EXECUTE privileges to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, the table owner has insufficient privileges to grant access on the type to other users.

> **See Also:** "Table Security" on page 25-5 for the requirements for creating a table

Privileges Required to Create Types and Tables Using Types Example

Assume that three users exist with the CONNECT and RESOURCE roles:

- USER1
- USER2
- USER3

USER1 performs the following DDL in his schema:

```
CREATE TYPE type1 AS OBJECT (
  attr1 NUMBER);

CREATE TYPE type2 AS OBJECT (
  attr2 NUMBER);

GRANT EXECUTE ON type1 TO user2;
GRANT EXECUTE ON type2 TO user2 WITH GRANT OPTION;
```

USER2 performs the following DDL in his schema:

```
CREATE TABLE tab1 OF user1.type1;
CREATE TYPE type3 AS OBJECT (
  attr3 user1.type2);
CREATE TABLE tab2 (
  col1 user1.type2);
```

The following statements succeed because USER2 has EXECUTE privilege on USER1's TYPE2 with the GRANT OPTION:

```
GRANT EXECUTE ON type3 TO user3;
GRANT SELECT on tab2 TO user3;
```

However, the following grant fails because USER2 does not have EXECUTE privilege on USER1's TYPE1 with the GRANT OPTION:

```
GRANT SELECT ON tab1 TO user3;
```

USER3 can successfully perform the following statements:

```
CREATE TYPE type4 AS OBJECT (
  attr4 user2.type3);
CREATE TABLE tab3 OF type4;
```

Privileges on Type Access and Object Access

Existing column-level and table-level privileges for DML commands apply to both column objects and row objects. Oracle9*i* defines the privileges shown in Table 25–2 for object tables:

Table 25–2 Privileges for Object Tables

Privilege	Allows you to...
SELECT	Access an object and its attributes from the table
UPDATE	Modify the attributes of the objects that make up the table's rows
INSERT	Create new objects in the table
DELETE	Delete rows

Similar table privileges and column privileges apply to column objects. Retrieving instances does not in itself reveal type information. However, clients must access named type information in order to interpret the type instance images. When a client requests such type information, Oracle checks for EXECUTE privilege on the type.

Consider the following schema:

```
CREATE TYPE emp_type (
    eno NUMBER, ename CHAR(31), eaddr addr_t);
CREATE TABLE emp OF emp_t;
```

and the following two queries:

```
SELECT VALUE(emp) FROM emp;
SELECT eno, ename FROM emp;
```

For either query, Oracle checks the user's SELECT privilege for the EMP table. For the first query, the user needs to obtain the EMP_TYPE type information to interpret the data. When the query accesses the EMP_TYPE type, Oracle checks the user's EXECUTE privilege.

Execution of the second query, however, does not involve named types, so Oracle does not check type privileges.

Additionally, using the schema from the previous section, USER3 can perform the following queries:

```
SELECT tab1.col1.attr2 FROM user2.tab1 tab1;
SELECT attr4.attr3.attr2 FROM tab3;
```

Note that in both SELECT statements, USER3 does not have explicit privileges on the underlying types, but the statement succeeds because the type and table owners have the necessary privileges with the GRANT OPTION.

Oracle checks privileges on the following events, and returns an error if the client does not have the privilege for the action:

- Pinning an object in the object cache using its REF value causes Oracle to check SELECT privilege on the containing object table.
- Modifying an existing object or flushing an object from the object cache causes Oracle to check UPDATE privilege on the destination object table.
- Flushing a new object causes Oracle to check INSERT privilege on the destination object table.
- Deleting an object causes Oracle to check DELETE privilege on the destination table.
- Pinning an object of named type causes Oracle to check EXECUTE privilege on the object.

Modifying an object's attributes in a client 3GL application causes Oracle to update the entire object. Hence, the user needs UPDATE privilege on the object table. UPDATE privilege on only certain columns of the object table is not sufficient, even if the application only modifies attributes corresponding to those columns. Therefore, Oracle does not support column level privileges for object tables.

Type Dependencies

As with stored objects such as procedures and tables, types being referenced by other objects are called dependencies. There are some special issues for types depended upon by tables. Because a table contains data that relies on the type definition for access, any change to the type causes all stored data to become inaccessible. Changes that can cause this effect are when necessary privileges required by the type are revoked or the type or dependent types are dropped. If either of these actions occur, then the table becomes invalid and cannot be accessed.

A table that is invalid because of missing privileges can automatically become valid and accessible if the required privileges are granted again. A table that is invalid because a dependent type has been dropped can never be accessed again, and the only permissible action is to drop the table.

Because of the severe effects which revoking a privilege on a type or dropping a type can cause, the SQL statements REVOKE and DROP TYPE by default implement a restrict semantics. This means that if the named type in either

statement has table or type dependents, then an error is received and the statement aborts. However, if the `FORCE` clause for either statement is used, the statement always succeeds, and if there are depended-upon tables, they are invalidated.

> **See Also:** *Oracle9i Database Reference* for details about using the `REVOKE`, `DROP TYPE`, and `FORCE` clauses

Introduction to Roles

Oracle provides for easy and controlled privilege management through roles. **Roles** are named groups of related privileges that you grant to users or other roles. Roles are designed to ease the administration of end-user system and schema object privileges. However, roles are not meant to be used for application developers, because the privileges to access schema objects within stored programmatic constructs need to be granted directly.

These properties of roles allow for easier privilege management within a database:

Reduced privilege administration	Rather than granting the same set of privileges explicitly to several users, you can grant the privileges for a group of related users to a role, and then only the role needs to be granted to each member of the group.
Dynamic privilege management	If the privileges of a group must change, only the privileges of the role need to be modified. The security domains of all users granted the group's role automatically reflect the changes made to the role.
Selective availability of privileges	You can selectively enable or disable the roles granted to a user. This allows specific control of a user's privileges in any given situation.
Application awareness	The data dictionary records which roles exist, so you can design applications to query the dictionary and automatically enable (or disable) selective roles when a user attempts to execute the application by way of a given username.
Application-specific security	You can protect role use with a password. Applications can be created specifically to enable a role when supplied the correct password. Users cannot enable the role if they do not know the password.

Database administrators often create roles for a database application. The DBA grants a secure application role all privileges necessary to run the application. The DBA then grants the secure application role to other roles or users. An application can have several different roles, each granted a different set of privileges that allow for more or less data access while using the application.

The DBA can create a role with a password to prevent unauthorized use of the privileges granted to the role. Typically, an application is designed so that when it starts, it enables the proper role. As a result, an application user does not need to know the password for an application's role.

See Also:

- "Data Definition Language Statements and Roles" on page 25-22 for information about restrictions for procedures
- *Oracle9i Application Developer's Guide - Fundamentals* for instructions for enabling roles from an application

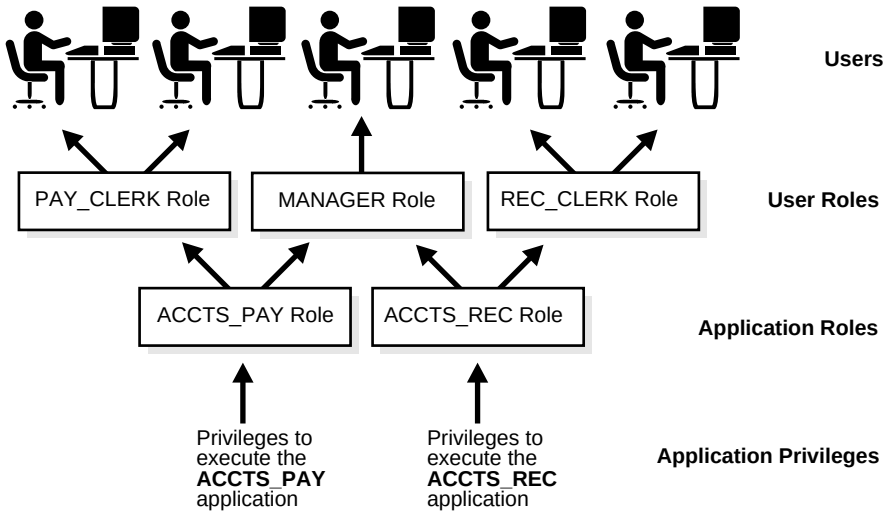
Common Uses for Roles

In general, you create a role to serve one of two purposes:

- To manage the privileges for a database application
- To manage the privileges for a user group

Figure 25–1 and the sections that follow describe the two uses of roles.

Figure 25–1 Common Uses for Roles



Secure Application Roles

You grant a secure application role all privileges necessary to run a given database application. Then, you grant the secure application role to other roles or to specific users. An application can have several different roles, with each role assigned a different set of privileges that allow for more or less data access while using the application.

User Roles

You create a user role for a group of database users with common privilege requirements. You manage user privileges by granting secure application roles and privileges to the user role and then granting the user role to appropriate users.

The Mechanisms of Roles

Database roles have the following functionality:

- A role can be granted system or schema object privileges.
- A role can be granted to other roles. However, a role cannot be granted to itself and cannot be granted circularly. For example, role A cannot be granted to role B if role B has previously been granted to role A.

- Any role can be granted to any database user.
- Each role granted to a user is, at a given time, either enabled or disabled. A user's security domain includes the privileges of all roles currently enabled for the user and excludes the privileges of any roles currently disabled for the user. Oracle allows database applications and users to enable and disable roles to provide selective availability of privileges.
- An indirectly granted role is a role granted to a role. It can be explicitly enabled or disabled for a user. However, by enabling a role that contains other roles, you implicitly enable all indirectly granted roles of the directly granted role.

Grant and Revoke Roles

You grant or revoke roles from users or other roles using the following options:

- The Grant System Privileges/Roles dialog box and Revoke System Privileges/Roles dialog box of Oracle Enterprise Manager
- The SQL statements `GRANT` and `REVOKE`

Privileges are granted to and revoked from roles using the same options. Roles can also be granted to and revoked from users using the operating system that executes Oracle, or through network services.

> **See Also:** *Oracle9i Database Administrator's Guide* for detailed instructions about role management

Who Can Grant or Revoke Roles?

Any user with the `GRANT ANY ROLE` system privilege can grant or revoke *any* role except a global role to or from other users or roles of the database. You should grant this system privilege conservatively because it is very powerful.

Any user granted a role with the `ADMIN OPTION` can grant or revoke that role to or from other users or roles of the database. This option allows administrative powers for roles on a selective basis.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about global roles

Role Names

Within a database, each role name must be unique, and no username and role name can be the same. Unlike schema objects, roles are not contained in any schema. Therefore, a user who creates a role can be dropped with no effect on the role.

Security Domains of Roles and Users

Each role and user has its own unique security domain. A role's security domain includes the privileges granted to the role plus those privileges granted to any roles that are granted to the role.

A user's security domain includes privileges on all schema objects in the corresponding schema, the privileges granted to the user, and the privileges of roles granted to the user that are **currently enabled**. (A role can be simultaneously enabled for one user and disabled for another.) A user's security domain also includes the privileges and roles granted to the user group `PUBLIC`.

PL/SQL Blocks and Roles

The use of roles in a PL/SQL block depends on whether it is an anonymous block or a named block (stored procedure, function, or trigger), and whether it executes with definer rights or invoker rights.

Named Blocks with Definer Rights

All roles are disabled in any named PL/SQL block (stored procedure, function, or trigger) that executes with definer rights. Roles are not used for privilege checking and you cannot set roles within a definer-rights procedure.

The `SESSION_ROLES` view shows all roles that are currently enabled. If a named PL/SQL block that executes with definer rights queries `SESSION_ROLES`, the query does not return any rows.

See Also: *Oracle9i Database Reference*

Anonymous Blocks with Invoker Rights

Named PL/SQL blocks that execute with invoker rights and anonymous PL/SQL blocks are executed based on privileges granted through enabled roles. Current roles are used for privilege checking within an invoker-rights PL/SQL block, and you can use dynamic SQL to set a role in the session.

See Also:

- *PL/SQL User's Guide and Reference* for an explanation of invoker and definer rights
- "Dynamic SQL in PL/SQL" on page 16-22

Data Definition Language Statements and Roles

A user requires one or more privileges to successfully execute a data definition language (DDL) statement, depending on the statement. For example, to create a table, the user must have the `CREATE TABLE` or `CREATE ANY TABLE` system privilege. To create a view of another user's table, the creator requires the `CREATE VIEW` or `CREATE ANY VIEW` system privilege and either the `SELECT` *`object`* privilege for the table or the `SELECT ANY TABLE` system privilege.

Oracle avoids the dependencies on privileges received by way of roles by restricting the use of specific privileges in certain DDL statements. The following rules outline these privilege restrictions concerning DDL statements:

- All system privileges and schema object privileges that permit a user to perform a DDL operation are usable when received through a role. For example:
  - System Privileges: the `CREATE TABLE`, `CREATE VIEW` and `CREATE PROCEDURE` privileges.
  - Schema Object Privileges: the `ALTER` and `INDEX` privileges for a table.

  Exception: The `REFERENCES` object privilege for a table cannot be used to define a table's foreign key if the privilege is received through a role.
- All system privileges and object privileges that allow a user to perform a DML operation that is required to issue a DDL statement are *not* usable when received through a role. For example:
  - A user who receives the `SELECT ANY TABLE` system privilege or the `SELECT` *`object`* privilege for a table through a role can use neither privilege to create a view on another user's table.

The following example further clarifies the permitted and restricted uses of privileges received through roles:

Assume that a user is:

- Granted a role that has the `CREATE VIEW` system privilege
- Granted a role that has the `SELECT` *`object`* privilege for the `EMP` table, but the user is indirectly granted the `SELECT` *`object`* privilege for the `EMP` table

- Directly granted the SELECT *object* privilege for the DEPT table

Given these directly and indirectly granted privileges:

- The user can issue SELECT statements on both the EMP and DEPT tables.
- Although the user has both the CREATE VIEW and SELECT privilege for the EMP table through a role, the user cannot create a usable view on the EMP table, because the SELECT *object* privilege for the EMP table was granted through a role. Any views created will produce errors when accessed.
- The user can create a view on the DEPT table, because the user has the CREATE VIEW privilege through a role and the SELECT privilege for the DEPT table directly.

Predefined Roles

The following roles are defined automatically for Oracle databases:

- CONNECT
- RESOURCE
- DBA
- EXP_FULL_DATABASE
- IMP_FULL_DATABASE

These roles are provided for backward compatibility to earlier versions of Oracle and can be modified in the same manner as any other role in an Oracle database.

The Operating System and Roles

In some environments, you can administer database security using the operating system. The operating system can be used to manage the granting (and revoking) of database roles and to manage their password authentication. This capability is not available on all operating systems.

> **See Also:** Your operating system specific Oracle documentation for details on managing roles through the operating system

Roles in a Distributed Environment

When you use roles in a distributed database environment, you must ensure that all needed roles are set as the default roles for a distributed (remote) session. You cannot enable roles when connecting to a remote database from within a local

database session. For example, you cannot execute a remote procedure that attempts to enable a role at the remote site.

See Also: *Oracle9i Heterogeneous Connectivity Administrator's Guide*

Fine-Grained Access Control

Fine-grained access control allows you to implement security policies with functions and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed (for example, by ad hoc queries).

You can:

- Use different policies for `SELECT, INSERT, UPDATE, and DELETE.`
- Use security policies only where you need them (for example, on salary information).
- Use more than one policy for each table, including building on top of base policies in packaged applications.
- Distinguish policies between different applications, by using *policy groups*. Each policy group indicates a set of policies that belong to an application.

Policy groups were introduced in Oracle9*i*, Release 1 (9.0.1). The database administrator designates an application context, called a *driving context*, to indicate the policy group in effect. When tables or views are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all the associated policies that belong to that policy group.

The PL/SQL package `DBMS_RLS` allows you to administer your security policies. Using this package, you can add, drop, enable, disable, and refresh the policies you create.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about package implementation

Dynamic Predicates

The function or package that implements the security policy you create returns a predicate (a `WHERE` condition). This predicate controls access as set out by the policy. Rewritten queries are fully optimized and shareable.

Security Policy Example

Consider the following security policy example.

In a human resources application called HR, EMPLOYEES is a view for the ALL_EMPLOYEES table and both objects are under the APPS schema. Following are the statements to create the table and the view:

```
CREATE TABLE all_employees
(employee_id  NUMBER(15),
 emp_name     VARCHAR2(30),
 mgr_id       NUMBER(15),
 user_name    VARCHAR2(8), .... );
CREATE VIEW employees AS SELECT * FROM all_employees;
```

You want to create a security policy function that limits access to the EMPLOYEES view, based on the user's role in the company. The predicates for the policy can be generated by the SECURE_PERSON function in the HR_ACCESS package. The package is under the schema APPS and contains functions to support all security policies related to the HR application. Also all the application contexts are under the APPS_SEC namespace. Following is the statement to create the application context for this example:

```
CREATE CONTEXT hr_role USING apps_sec.hr_role
```

Following are the statements to create the security policy function:

```
CREATE PACKAGE BODY hr_access IS
    FUNCTION secure_person(obj_schema VARCHAR2, obj_name VARCHAR2)
             RETURN VARCHAR2 IS
        d_predicate VARCHAR2(2000);
    BEGIN
        IF SYS_CONTEXT ('apps_sec', 'hr_role') = 'EMP' THEN
            d_predicate = 'emp_name = sys_context(''userenv'', ''user'')';
        IF SYS_CONTEXT ('apps_sec', 'hr_role') = 'MGR' THEN
            d_predicate = 'mgr_id = sys_context(''userenv'', ''uid''))';
        ELSE
             d_predicate = '1=2';  -- deny access to other users,
                                   -- may use something like 'keycol=null'
        RETURN d_predicate;
    END secure_person;
END hr_access;
```

The next step is to associate a policy called PER_PEOPLE_SEC for the EMPLOYEES view to the HR_ACCESS.SECURE_PERSON function that generates the dynamic predicates:

```
DBMS_RLS.ADD_POLICY('apps', 'employees', 'per_people_sec', 'apps'
                    'hr_access.secure_person',  'select, update, delete');
```

Now any `SELECT, UPDATE,` and `DELETE` statement with the `EMPLOYEES` view involved will pick up one of the three predicates based on the value of the application context `HR_ROLE.`

Note that the same security policy function that secured the `ALL_EMPLOYEES` table can also be used to generate the dynamic predicates to secure the `ADDRESSES` table because they have the same policy to limit access to data.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for details about establishing security policies

Application Context

Application context facilitates the implementation of fine-grained access control. It allows you to implement security policies with functions and then associate those security policies with applications. Each application can have its own application-specific context. Users are not allowed to arbitrarily change their context (for example, through SQL*Plus).

Application contexts permit flexible, parameter-based access control, based on attributes of interest to an application. For example, context attributes for a human resources application could include "position," "organizational unit," and "country," whereas attributes for an order-entry control might be "customer number" and "sales region".

You can:

- Base predicates on context values
- Use context values within predicates, as bind variables
- Set user attributes
- Access user attributes

To define an application context:

1. Create a PL/SQL package with functions that validate and set the context for your application. You may want to use an event trigger on login to set the initial context for logged-in users.
2. Use `CREATE CONTEXT` to specify a unique context name and associate it with the PL/SQL package you created.

3. Do one of the following:
   - Reference the application context in a policy function implementing fine-grained access control.
   - Create an event trigger on login to set the initial context for a user. For example, you could query a user's employee number and set this as an "employee number" context value.
4. Reference the application context.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i Application Developer's Guide - Fundamentals*

Secure Application Roles

Oracle9*i*, Release 1 (9.0.1), provides secure application roles, roles that can be enabled only by authorized PL/SQL packages. This mechanism restricts the enabling of roles to the invoking application.

In previous releases, passwords were either embedded in the source code or stored in a table. Application developers no longer need to secure a role by embedding passwords inside applications. Instead, they create a secure application role and specify which PL/SQL package is authorized to enable the role. Package identity is used to determine whether there are sufficient privileges to enable the roles. The application performs authentication before enabling the role.

The application can perform customized authorization, such as checking whether the user has connected through a proxy, before enabling the role.

Note: Because of the restriction that users can not change security domain inside Definer's Right procedures, secure application roles can only be enabled inside Invoker's Right procedures.

Creation of Secure Application Roles

Secure application roles are created by using the `CREATE ROLE ... IDENTIFIED USING` statement. Here is an example:

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

This indicates the following:

- The role `admin_role` to be created is a secure application role.
- The role can only be enabled by any module defined inside the PL/SQL package `hr.admin`.

You must have the system privilege `CREATE ROLE` to execute this command.

Roles that are enabled inside an Invoker's Right procedure remain in effect even after the procedure exits. Therefore, you can have a dedicated procedure that deals with enabling the role for the rest of the session to use.

See Also:

- *Oracle9i SQL Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i Application Developer's Guide - Fundamentals*

26

Auditing

This chapter discusses the auditing feature of Oracle. It includes:

- Introduction to Auditing
- Statement Auditing
- Privilege Auditing
- Schema Object Auditing
- Fine-Grained Auditing
- Focus Statement, Privilege, and Schema Object Auditing
- Audit in a Multitier Environment

Introduction to Auditing

Auditing is the monitoring and recording of selected user database actions. Auditing is normally used to:

- Investigate suspicious activity. For example, if an unauthorized user is deleting data from tables, the security administrator might decide to audit all connections to the database and all successful and unsuccessful deletions of rows from all tables in the database.
- Monitor and gather data about specific database activities. For example, the database administrator can gather statistics about which tables are being updated, how many logical I/Os are performed, or how many concurrent users connect at peak times.

Features of Auditing

This section outlines the features of the Oracle auditing mechanism.

Types of Auditing

Oracle supports three general types of auditing:

Statement auditing	The selective auditing of SQL statements with respect to only the type of statement, not the specific schema objects on which it operates. Statement auditing options are typically broad, auditing the use of several types of related actions for each option. For example, `AUDIT TABLE` tracks several DDL statements regardless of the table on which they are issued. You can set statement auditing to audit selected users or every user in the database.
Privilege auditing	The selective auditing of the use of powerful system privileges to perform corresponding actions, such as `AUDIT CREATE TABLE.` Privilege auditing is more focused than statement auditing because it audits only the use of the target privilege. You can set privilege auditing to audit a selected user or every user in the database.
Schema object auditing	The selective auditing of specific statements on a particular schema object, such as `AUDIT SELECT ON EMP.` Schema object auditing is very focused, auditing only a specific statement on a specific schema object. Schema object auditing always applies to all users of the database.

Fine-grained auditing Fine-grained auditing allows the monitoring of data access based on content.

Focus of Auditing

Oracle allows audit options to be focused or broad. You can audit:

- Successful statement executions, unsuccessful statement executions, or both
- Statement executions once in each user session or once every time the statement is executed
- Activities of all users or of a specific user

Audit Records and the Audit Trail

Audit records include information such as the operation that was audited, the user performing the operation, and the date and time of the operation. Audit records can be stored in either a data dictionary table, called the **database audit trail**, or an operating system audit trail.

The database audit trail is a single table named `FGA_LOG$` in the `SYS` schema of each Oracle database's data dictionary. Several predefined views are provided to help you use the information in this table.

The audit trail records can contain different types of information, depending on the events audited and the auditing options set. The following information is always included in each audit trail record, if the information is meaningful to the particular audit action:

- The user name
- The session identifier
- The terminal identifier
- The name of the schema object accessed
- The operation performed or attempted
- The completion code of the operation
- The date and time stamp
- The system privileges used

The operating system audit trail is encoded and not readable, but it is decoded in data dictionary files and error messages as follows:

Action code This describes the operation performed or attempted. The `AUDIT_ACTIONS` data dictionary table contains a list of these codes and their descriptions.

Privileges used This describes any system privileges used to perform the operation. The `SYSTEM_PRIVILEGE_MAP` table lists all of these codes and their descriptions.

Completion code This describes the result of the attempted operation. Successful operations return a value of zero, and unsuccessful operations return the Oracle error code describing why the operation was unsuccessful.

See Also:

- *Oracle9i Database Administrator's Guide* for instructions for creating and using predefined views
- *Oracle9i Database Error Messages* for a list of completion codes

Mechanisms for Auditing

This section explains the mechanisms used by the Oracle auditing features.

When Are Audit Records Generated?

The recording of audit information can be enabled or disabled. This functionality allows any authorized database user to set audit options at any time but reserves control of recording audit information for the security administrator.

When auditing is enabled in the database, an audit record is generated during the execute phase of statement execution.

SQL statements inside PL/SQL program units are individually audited, as necessary, when the program unit is executed.

The generation and insertion of an audit trail record is independent of a user's transaction. Therefore, even if a user's transaction is rolled back, the audit trail record remains committed.

Note: Audit records are never generated by sessions established by the user SYS or connections with administrator privileges. Connections by these users bypass certain internal features of Oracle to allow specific administrative operations to occur (for example, database startup, shutdown, recovery, and so on).

See Also:

- *Oracle9i Database Administrator's Guide* for instructions on enabling and disabling auditing
- Chapter 16, "SQL, PL/SQL, and Java" for information about the different phases of SQL statement processing and shared SQL

Events Always Audited to the Operating System Audit Trail

Regardless of whether database auditing is enabled, Oracle always records some database-related actions into the operating system audit trail:

Instance startup	An audit record is generated that details the operating system user starting the instance, the user's terminal identifier, the date and time stamp, and whether database auditing was enabled or disabled. This information is recorded into the operating system audit trail because the database audit trail is not available until after startup has successfully completed. Recording the state of database auditing at startup further prevents an administrator from restarting a database with database auditing disabled so that they are able to perform unaudited actions.
Instance shutdown	An audit record is generated that details the operating system user shutting down the instance, the user's terminal identifier, the date and time stamp.
Connections with administrator privileges	An audit record is generated that details the operating system user connecting to Oracle with administrator privileges. This provides accountability of users connected with administrator privileges.

On operating systems that do not make an audit trail accessible to Oracle, these audit trail records are placed in an Oracle audit trail file in the same directory as background process trace files.

> **See Also:** Your operating system specific Oracle documentation for more information about the operating system audit trail

When Do Audit Options Take Effect?

Statement and privilege audit options in effect at the time a database user connects to the database remain in effect for the duration of the session. A session does not see the effects of statement or privilege audit options being set or changed. The modified statement or privilege audit options take effect only when the current session is ended and a new session is created. In contrast, changes to schema object audit options become effective for current sessions immediately.

Audit in a Distributed Database

Auditing is site autonomous. An instance audits only the statements issued by directly connected users. A local Oracle node cannot audit actions that take place in a remote database. Because remote connections are established through the user account of a database link, the remote Oracle node audits the statements issued through the database link's connection.

> **See Also:** *Oracle9i Database Administrator's Guide*

Audit to the Operating System Audit Trail

Oracle allows audit trail records to be directed to an operating system audit trail if the operating system makes such an audit trail available to Oracle. On some other operating systems, these audit records are written to a file outside the database, with a format similar to other Oracle trace files.

> **See Also:** Your operating system specific Oracle documentation, to see if this feature has been implemented on your operating system

Oracle allows certain actions that are *always* audited to continue, even when the operating system audit trail (or the operating system file containing audit records) is unable to record the audit record. The usual cause of this is that the operating system audit trail or the file system is full and unable to accept new records.

System administrators configuring operating system auditing should ensure that the audit trail or the file system does not fill completely. Most operating systems

provide administrators with sufficient information and warning to ensure this does not occur. Note, however, that configuring auditing to use the database audit trail removes this vulnerability, because the Oracle server prevents audited events from occurring if the audit trail is unable to accept the database audit record for the statement.

Statement Auditing

Statement auditing is the selective auditing of related groups of statements that fall into two categories:

- DDL statements, regarding a particular type of database structure or schema object, but not a specifically named structure or schema object (for example, `AUDIT TABLE` audits all `CREATE` and `DROP TABLE` statements)
- DML statements, regarding a particular type of database structure or schema object, but not a specifically named structure or schema object (for example, `AUDIT SELECT TABLE` audits all `SELECT ... FROM TABLE/VIEW` statements, regardless of the table or view)

Statement auditing can be broad or focused, auditing the activities of all database users or the activities of only a select list of database users.

Privilege Auditing

Privilege auditing is the selective auditing of the statements allowed using a system privilege. For example, auditing of the `SELECT ANY TABLE` system privilege audits users' statements that are executed using the `SELECT ANY TABLE` system privilege. You can audit the use of any system privilege.

In all cases of privilege auditing, owner privileges and schema object privileges are checked before system privileges. If the owner and schema object privileges suffice to permit the action, the action is not audited.

If similar statement and privilege audit options are both set, only a single audit record is generated. For example, if the statement clause `TABLE` and the system privilege `CREATE TABLE` are both audited, only a single audit record is generated each time a table is created.

Privilege auditing is more focused than statement auditing because each option audits only specific types of statements, not a related list of statements. For example, the statement auditing clause `TABLE` audits `CREATE TABLE, ALTER TABLE,` and `DROP TABLE` statements, while the privilege auditing option

CREATE TABLE audits only CREATE TABLE statements. This is because only the CREATE TABLE statement requires the CREATE TABLE privilege.

Like statement auditing, privilege auditing can audit the activities of all database users or the activities of a select list of database users.

Schema Object Auditing

Schema object auditing is the selective auditing of specific DML statements (including queries) and GRANT and REVOKE statements for specific schema objects. Schema object auditing audits the operations permitted by schema object privileges, such as SELECT or DELETE statements on a given table, as well as the GRANT and REVOKE statements that control those privileges.

You can audit statements that reference tables, views, sequences, **standalone** stored procedures and functions, and packages. Procedures in packages cannot be audited individually.

Statements that reference clusters, database links, indexes, or synonyms are not audited directly. However, you can audit access to these schema objects indirectly by auditing the operations that affect the base table.

Schema object audit options are always set for all users of the database. These options cannot be set for a specific list of users. You can set default schema object audit options for all auditable schema objects.

> **See Also:** *Oracle9i SQL Reference* for information about auditable schema objects

Schema Object Audit Options for Views and Procedures

Views and procedures (including stored functions, packages, and triggers) reference underlying schema objects in their definition. Therefore, auditing with respect to views and procedures has several unique characteristics. Multiple audit records can be generated as the result of using a view or a procedure: The use of the view or procedure is subject to enabled audit options, and the SQL statements issued as a result of using the view or procedure are subject to the enabled audit options of the base schema objects (including default audit options).

Consider the following series of SQL statements:

```
AUDIT SELECT ON emp;

CREATE VIEW emp_dept AS
  SELECT empno, ename, dname
```

```
    FROM emp, dept
    WHERE emp.deptno = dept.deptno;

AUDIT SELECT ON emp_dept;

SELECT * FROM emp_dept;
```

As a result of the query on EMP_DEPT, two audit records are generated: one for the query on the EMP_DEPT view and one for the query on the base table EMP (indirectly through the EMP_DEPT view). The query on the base table DEPT does not generate an audit record because the SELECT audit option for this table is not enabled. All audit records pertain to the user that queried the EMP_DEPT view.

The audit options for a view or procedure are determined when the view or procedure is first used and placed in the shared pool. These audit options remain set until the view or procedure is flushed from, and subsequently replaced in, the shared pool. Auditing a schema object invalidates that schema object in the cache and causes it to be reloaded. Any changes to the audit options of base schema objects are not observed by views and procedures in the shared pool.

Continuing with the previous example, if auditing of SELECT statements is turned off for the EMP table, use of the EMP_DEPT view no longer generates an audit record for the EMP table.

Fine-Grained Auditing

Oracle9*i*, Release 1 (9.0.1), provides fine-grained auditing, which allows the monitoring of data access based on content. More importantly, the monitoring does not depend on how it is done. A built-in audit mechanism in the database prevents users from by-passing the audit. Oracle DBMS has already provided triggers capability for potentially monitoring DML actions such as INSERT/UPDATE/DELETE. However, monitoring on SELECT is costly and may not work for certain cases. In addition, users may want to define their own alert action in addition to just inserting an audit record into the audit trail. This feature provides an extensible interface to audit SELECT statements on tables and views.

A PLSQL package, DBMS_FGA, administers these value-based audit policies. Using DBMS_FGA, the security administrator creates an audit policy on the target table. If any of the rows returned from a query block matches the audit condition (these rows are referred to as **interested** rows), an audit event entry, including username, SQL text, bind variable, policy name, session id, time stamp, and other attributes, is inserted into the audit trail. As part of the extensibility framework, administrators can also optionally define an appropriate event handler, an **audit**

event handler, to process the event; for example, the audit event handler could send an alert page to the administrator.

For example, to audit `SELECT` statements on table `hr.emp` to monitor any query that accesses the `salary` column of the employee records that belong to the `sales` department, the administrator can issue the following SQL statement to set up the auditing:

```
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary');
```

After the auditing is set up, the following SQL statements will cause the database to log an audit event record:

```
SELECT count(*) FROM hr.emp WHERE dept = 'SALES' and salary > 10000000;
or
SELECT salary FROM hr.emp WHERE dept = 'SALES';
```

All the relevant information has been supplied, along with a trigger-like mechanism; administrators need only define what to record and how to process the audit event. For example, suppose the following command has been issued:

```
/* create audit event handler */
CREATE PROCEDURE sec.log_id (schema varchar2, table varchar2, policy varchar2)
AS
luser varchar2(30);
sql varchar2(2000);
time date;
BEGIN
UTIL_ALERT_PAGER(schema, table, policy);      -- send an alert note to my pager
END;


/* add the policy */
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary',
handler_schema => 'sec',
event_handler_module => 'log_id',
```

```
enable              => TRUE);
```

After the fetch of the first interested row, the event is recorded, and the audit function `sec.log_id` is fired. The audit event record that is generated gets stored in the new format of `fga_log$,` which has reserved columns for recording SQL text, bind variables and policy name.

See Also: *Oracle9i Application Developer's Guide - Fundamentals*

Focus Statement, Privilege, and Schema Object Auditing

Oracle allows you to focus statement, privilege, and schema object auditing in three areas:

- Successful and unsuccessful executions of the audited SQL statement
- `BY SESSION` and `BY ACCESS` auditing
- For specific users or for all users in the database (statement and privilege auditing only)

Successful and Unsuccessful Statement Executions Auditing

For statement, privilege, and schema object auditing, Oracle allows the selective auditing of successful executions of statements, unsuccessful attempts to execute statements, or both. Therefore, you can monitor actions even if the audited statements do not complete successfully.

You can audit an unsuccessful statement execution only if a valid SQL statement is issued but fails because of lack of proper authorization or because it references a nonexistent schema object. Statements that failed to execute because they simply were not valid cannot be audited. For example, an enabled privilege auditing option set to audit unsuccessful statement executions audits statements that use the target system privilege but have failed for other reasons (such as when `CREATE TABLE` is set but a `CREATE TABLE` statement fails due to lack of quota for the specified tablespace).

Using either form of the `AUDIT` statement, you can include:

- The `WHENEVER SUCCESSFUL` clause, to audit only successful executions of the audited statement
- The `WHENEVER NOT SUCCESSFUL` clause, to audit only unsuccessful executions of the audited statement

- Neither of the previous clauses, to audit both successful and unsuccessful executions of the audited statement

BY SESSION compared with BY ACCESS Clauses of Audit Statement

Most auditing options can be set to indicate how audit records should be generated if the audited statement is issued multiple times in a single user session. This section describes the distinction between the `BY SESSION` and `BY ACCESS` clauses of the `AUDIT` statement.

See Also: *Oracle9i SQL Reference*

BY SESSION

For any type of audit (schema object, statement, or privilege), `BY SESSION` inserts only one audit record in the audit trail, for each user and schema object, during the session that includes an audited action.

A **session** is the time between when a user connects to and disconnects from an Oracle database.

BY SESSION Example 1 Assume the following:

- The `SELECT TABLE` statement auditing option is set `BY SESSION.`
- `JWARD` connects to the database and issues five `SELECT` statements against the table named `DEPT` and then disconnects from the database.
- `SWILLIAMS` connects to the database and issues three `SELECT` statements against the table `EMP` and then disconnects from the database.

In this case, the audit trail contains two audit records for the eight `SELECT` statements— one for each session that issued a `SELECT` statement.

BY SESSION Example 2 Alternatively, assume the following:

- The `SELECT TABLE` statement auditing option is set `BY SESSION.`
- `JWARD` connects to the database and issues five `SELECT` statements against the table named `DEPT,` and three `SELECT` statements against the table `EMP,` and then disconnects from the database.

In this case, the audit trail contains two records—one for each schema object against which the user issued a `SELECT` statement in a session.

Note: If you use the `BY SESSION` clause when directing audit records to the operating system audit trail, Oracle generates and stores an audit record each time an access is made. Therefore, in this auditing configuration, `BY SESSION` is equivalent to `BY ACCESS.`

BY ACCESS

Setting audit `BY ACCESS` inserts one audit record into the audit trail for each execution of an auditable operation within a cursor. Events that cause cursors to be reused include the following:

- An application, such as Oracle Forms, holding a cursor open for reuse
- Subsequent execution of a cursor using new bind variables
- Statements executed within PL/SQL loops where the PL/SQL engine optimizes the statements to reuse a single cursor

Note that auditing is **not** affected by whether a cursor is shared. Each user creates her or his own audit trail records on first execution of the cursor.

For example, assume that:

- The `SELECT TABLE` statement auditing option is set `BY ACCESS.`
- `JWARD` connects to the database and issues five `SELECT` statements against the table named `DEPT` and then disconnects from the database.
- `SWILLIAMS` connects to the database and issues three `SELECT` statements against the table `DEPT` and then disconnects from the database.

The single audit trail contains eight records for the eight `SELECT` statements.

Defaults and Excluded Operations

The AUDIT statement allows you to specify either BY SESSION or BY ACCESS. However, several audit options can be set only BY ACCESS, including:

- All statement audit options that audit DDL statements
- All privilege audit options that audit DDL statements

For all other audit options, BY SESSION is used by default.

Audit By User

Statement and privilege audit options can audit statements issued by any user or statements issued by a specific list of users. By focusing on specific users, you can minimize the number of audit records generated.

Audit By User Example To audit statements by the users SCOTT and BLAKE that query or update a table or view, issue the following statements:

```
AUDIT SELECT TABLE, UPDATE TABLE
     BY scott, blake;
```

> **See Also:** *Oracle9i SQL Reference* for more information about auditing by user

Audit in a Multitier Environment

In a multitier environment, Oracle preserves the identity of a client through all tiers. This enables auditing of actions taken on behalf of the client. To do so, use the BY **proxy** clause in your AUDIT statement.

This clause allows you a few options. You can:

- Audit SQL statements issued by the specific proxy on its own behalf
- Audit statements executed on behalf of a specified user or users
- Audit all statements executed on behalf of any user

The middle tier can set the light-weight user identity in a database session so that it will show up in audit trail. You use OCI or PL/SQL to set the client identifier.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle Call Interface Programmer's Guide*
- *PL/SQL User's Guide and Reference*

A

Operating System Specific Information

This manual occasionally refers to other Oracle manuals that contain detailed information for using Oracle on a specific operating system. These Oracle manuals are often called **installation and configuration guides**, although the exact name can vary on different operating systems.

This appendix lists all the references in this manual to operating system specific Oracle manuals, and lists the operating system-dependent initialization parameters. If you are using Oracle on multiple operating systems, this appendix can help you ensure that your applications are portable across these operating systems.

Operating system specific topics in this manual are listed alphabetically as follows.

- Administrator privileges, prerequisites: "Connection with Administrator Privileges" on page 6-3; connect string syntax: *Oracle9i Net Services Administrator's Guide*
- Auditing: "Events Always Audited to the Operating System Audit Trail" on page 26-5 and "Audit to the Operating System Audit Trail" on page 26-6
- Authenticating DBAs: "Connection with Administrator Privileges" on page 6-3 and "Authentication of Database Administrators" on page 24-14
- Authenticating users: "Authentication by the Operating System" on page 24-15
- Background processes, ARC*n*: "Archiver Processes (ARCn)" on page 9-13 and the *Oracle9i Backup and Recovery Concepts*
- Background processes, creating: "Background Processes" on page 9-5
- Background processes, DBW*n* processes: "Database Writer Process (DBWn)" on page 9-8
- Client / server communication: "Dedicated Server Configuration" on page 9-21
- Communication software: "Communications Software for the Operating System" on page 9-24
- Configuring Oracle: "Types of Processes" on page 9-2; for dedicated server (two-task Oracle), see "Dedicated Server Configuration" on page 9-21; for multithreaded server, see "Shared Server Architecture" on page 9-15
- Data blocks, size of: "Data Blocks Overview" on page 3-3
- Datafiles, size of file header: "Datafiles Overview" on page 4-18
- Dedicated server, requesting for administrative operations: "Restricted Operations of the Shared Server" on page 9-20
- Indexes, overhead of index blocks: "Format of Index Blocks" on page 11-33
- Operating system audit trails: On operating systems that do not make an audit trail accessible to Oracle, these audit trail records are placed in an Oracle audit trail file in the same directory as background process trace files.

  Oracle allows audit trail records to be directed to an operating system audit trail if the operating system makes such an audit trail available to Oracle. On some other operating systems, these audit records are written to a file outside the database, with a format similar to other Oracle trace files.

Check your platform-specific Oracle documentation to see if this feature has been implemented on your operating system.

- Oracle Net Services, choosing and installing network drivers: "Program Interface Drivers" on page 9-23
- Oracle Net Services, drivers included in Oracle Net Services software: "How Oracle Net Services Works" on page 7-9; also, the *Oracle9i Net Services Administrator's Guide*
- Password files and authentication schemes: "Authentication of Database Administrators" on page 24-14
- Program global areas (PGAs): "SQL Work Areas" on page 8-18
- Role management by the operating system: "The Operating System and Roles" on page 25-23
- Rollback segments, number of transactions in each: "Transactions and Rollback Segments" on page 3-26
- Software code areas, shared or unshared: "Software Code Areas" on page 8-21

Glossary

AFTER triggers

When defining a trigger, you can specify the trigger timing—whether the trigger action is to be executed before or after the triggering statement.

`AFTER` triggers execute the trigger action after the triggering statement is executed.

`BEFORE` and `AFTER` apply to both statement and row triggers.

See Also: trigger

architecture

See: Oracle architecture

archived redo log

Optionally, filled online redo files can be archived before being reused, creating an archived redo log. Archived (offline) redo log files constitute the archived redo log.

See Also: redo log

background processes

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

Oracle creates a set of background processes for each instance.

See Also: instance, process, Oracle processes, user processes

BEFORE triggers

When defining a trigger, you can specify the trigger timing—whether the trigger action is to be executed before or after the triggering statement.

`BEFORE` triggers execute the trigger action before the triggering statement is executed.

`BEFORE` and `AFTER` apply to both statement and row triggers.

See Also: trigger

buffer cache

The database buffer cache is the portion of the SGA that holds copies of data blocks read from data files. All user processes concurrently connected to the instance share access to the database buffer cache.

See Also: system global area (SGA)

byte semantics

The length of string is measured in bytes.

character semantics

The length of string is measured in characters.

CHECK constraints

A `CHECK` integrity constraint on a column or set of columns requires that a specified condition be true or unknown for every row of the table. If a DML statement results in the condition of the `CHECK` constraint evaluating to false, then the statement is rolled back.

client

In client/server architecture, the front-end database application, which interacts with a user through the keyboard, display, and pointing device such as a mouse. The client portion has no data access responsibilities. It concentrates on requesting, processing, and presenting data managed by the server portion.

See Also: client/server architecture, server

client/server architecture

Software architecture based on a separation of processing between two CPUs, one acting as the client in the transaction, requesting and receiving services, and the other as the server that provides services in a transaction.

cluster

Optional structure for storing table data. Clusters are groups of one or more tables physically stored together because they share common columns and are often used together. Because related rows are physically stored together, disk access time improves.

column

Vertical space in a database table that represents a particular domain of data. A column has a column name and a specific datatype. For example, in a table of employee information, all of the employees' dates of hire would constitute one column.

See Also: row, table

commit

Make permanent changes to data (inserts, updates, deletes) in the database. Before changes are committed, both the old and new data exist so that changes can be stored or the data can be restored to its prior state.

See Also: roll back

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

connection

Communication pathway between a user process and an Oracle instance.

See Also: session, user processes

control file

A file that records the physical structure of a database and contains the database name, the names and locations of associated databases and online redo log files, the time stamp of the database creation, the current log sequence number, and checkpoint information.

See Also: physical structures, redo log

database

Collection of data that is treated as a unit. The purpose of a database is to store and retrieve related information.

database buffer

One of several types of memory structures that stores information within the system global area. Database buffers store the most recently used blocks of data.

See Also: system global area (SGA)

database buffer cache

Memory structure in the system global area that stores the most recently used blocks of data.

See Also: system global area (SGA)

database link

A named schema object that describes a path from one database to another. Database links are implicitly used when a reference is made to a global object name in a distributed database.

database writer process (DBW*n*)

An Oracle background process that writes the contents of buffers to data files. The DBW*n* processes are responsible for writing modified (dirty) buffers in the database buffer cache to disk.

See Also: buffer cache

data block

Smallest logical unit of data storage in an Oracle database. Also called logical blocks, Oracle blocks, or pages. One data block corresponds to a specific number of bytes of physical database space on disk.

See Also: extent, segment

data consistency

In a multiuser environment, where many users can access data at the same time (concurrency), data consistency means that each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users.

See Also: concurrency

data dictionary

The central set of tables and views that are used as a read-only reference about a particular database. A data dictionary stores such information as:

- The logical and physical structure of the database
- Valid users of the database
- Information about integrity constraints
- How much space is allocated for a schema object and how much of it is in use

A data dictionary is created when a database is created and is automatically updated when the structure of the database is updated.

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 the entry of invalid information into tables.

See Also: integrity constraint, trigger

data file

A physical operating system file on disk that was created by Oracle and contains data structures such as tables and indexes. A data file can only belong to one database.

See Also: indexes, physical structures

data segment

Each nonclustered table has a data segment. All of the table's data is stored in the extents of its data segment. For a partitioned table, each partition has a data segment.

Each cluster has a data segment. The data of every table in the cluster is stored in the cluster's data segment.

See Also: cluster, extent, segment

DBTZ

Database time zone.

dedicated server

A database server configuration in which a server process handles requests for a single user process.

See Also: shared server

dispatcher processes (D*nnn*)

Optional background processes, present only when a shared server configuration is used. At least one dispatcher process is created for every communication protocol in use (D000, . . ., D*nnn*). Each dispatcher process is responsible for routing requests from connected user processes to available shared server processes and returning the responses back to the appropriate user processes.

See Also: shared server

distributed processing

Software architecture that uses more than one computer to divide the processing for a set of related jobs. Distributed processing reduces the processing load on a single computer.

DDL

Data Definition Language. Includes statements like `CREATE/ALTER TABLE/INDEX`, which define or change data structure.

DML

Data Manipulation Language. Includes statements like `INSERT`, `UPDATE`, and `DELETE`, which change data in tables.

DOP

The degree of parallelism of an operation.

extent

Second level of logical database storage. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.

See Also: data block, segment

foreign key

Integrity constraint that requires each value in a column or set of columns to match a value in a related table's `UNIQUE` or `PRIMARY KEY`.

`FOREIGN KEY` integrity constraints also define referential integrity actions that dictate what Oracle should do with dependent data if the data it references is altered.

See Also: integrity constraint, primary key

indexes

Optional structures associated with tables and clusters. You can create indexes on one or more columns of a table to speed SQL statement execution on that table.

See Also: cluster

index segment

Each index has an index segment that stores all of its data. For a partitioned index, each partition has an index segment.

See Also: indexes, segment

indextype

An object that registers a new indexing scheme by specifying the set of supported operators and routines that manage a domain index.

instance

A system global area (SGA) and the Oracle background processes constitute an Oracle instance. Every time a database is started, a system global area is allocated and Oracle background processes are started. The SGA is deallocated when the instance shuts down.

See Also: background processes, system global area (SGA)

integrity

See: data integrity

integrity constraint

Declarative method of defining a rule for a column of a table. Integrity constraints enforce the business rules associated with a database and prevent the entry of invalid information into tables.

key

Column or set of columns included in the definition of certain types of integrity constraints. Keys describe the relationships between the different tables and columns of a relational database.

See Also: integrity constraint, foreign key, primary key

large pool

Optional area in the system global area that provides large memory allocations for Oracle backup and restore operations, I/O server processes, and session memory for the shared server and Oracle XA.

See Also: system global area (SGA), process, shared server, Oracle XA

log writer process (LGWR)

The log writer process (LGWR) is responsible for redo log buffer management—writing the redo log buffer to a redo log file on disk. LGWR writes all redo entries that have been copied into the buffer since the last time it wrote.

See Also: redo log

logical structures

Logical structures of an Oracle database include tablespaces, schema objects, data blocks, extents, and segments. Because the physical and logical structures are separate, the physical storage of data can be managed without affecting the access to logical storage structures.

See Also: physical structures

materialized view

A materialized view provides indirect access to table data by storing the results of a query in a separate schema object.

See Also: view

NOT NULL constraint

Data integrity constraint that requires a column of a table contain no null values.

See Also: NULL value

NULL value

Absence of a value in a column of a row. Nulls indicate missing, unknown, or inapplicable data. A null should not be used to imply any other value, such as zero.

object type

An object type consists of two parts: a spec and a body. The type body always depends on its type spec.

offline redo log

See: archived redo log

online redo log

The online redo log is a set of two or more online redo log files that record all changes made to the database, including both uncommitted and committed changes. Redo entries are temporarily stored in redo log buffers of the system global area, and the background process LGWR writes the redo entries sequentially to an online redo log file.

See Also: redo log, system global area (SGA), background processes, log writer process (LGWR)

operator

In memory management, the term operator refers to a data flow operator, such as a sort, hash join, or bitmap merge.

Oracle architecture

Memory and process structures used by an Oracle server to manage a database.

See Also: database, process, server

Oracle processes

Oracle processes execute the Oracle server code. They include server processes and background processes.

See Also: process, server processes, background processes, user processes

Oracle XA

The Oracle XA library is an external interface that allows global transactions to be coordinated by a transaction manager other than the Oracle server.

physical structures

Physical database structures of an Oracle database include data files, redo log files, and control files.

See Also: logical structures

PL/SQL

Oracle's procedural language extension to SQL. PL/SQL enables you to mix SQL statements with procedural constructs. With PL/SQL, you can define and execute PL/SQL program units such as procedures, functions, and packages.

See Also: SQL

primary key

The column or set of columns included in the definition of a table's `PRIMARY KEY` constraint. A primary key's values uniquely identify the rows in a table. Only one primary key can be defined for each table.

See Also: PRIMARY KEY constraint

PRIMARY KEY constraint

Integrity constraint that disallows duplicate values and nulls in a column or set of columns.

See Also: integrity constraint

priority inversion

Priority inversion occurs when a high priority job is executed with lower amount of resources than a low priority job. Thus the expected priority is "inverted."

process

Each process in an Oracle instance performs a specific job. By dividing the work of Oracle and database applications into several processes, multiple users and applications can connect to a single database instance simultaneously.

See Also: Oracle processes, user processes

program global area (PGA)

A memory buffer that contains data and control information for a server process. A PGA is created by Oracle when a server process is started. The information in a PGA depends on the Oracle configuration.

query block

A self-contained DML against a table. A query block can be a top-level DML or a subquery.

See Also: DML

read consistency

In a multiuser environment, Oracle's read consistency ensures that

- The set of data seen by a statement remains constant throughout statement execution (statement-level read consistency).

- Readers and writer of database data do not wait for other writers or other readers of the same data. Writers of database data wait only for other writers who are updating identical rows in concurrent transactions.

See Also: concurrency, data consistency

Real Application Clusters

Option with Oracle9*i* Enterprise Edition that allows multiple concurrent instances to share a single physical database.

See Also: instance

redo log

A set of files that protect altered database data in memory that has not been written to the datafiles. The redo log can consist of two parts: the online redo log and the archived redo log.

See Also: online redo log, archived redo log

redo log buffer

Memory structure in the system global area that stores redo entries—a log of changes made to the database. The redo entries stored in the redo log buffers are written to an online redo log file, which is used if database recovery is necessary.

See Also: system global area (SGA)

referential integrity

A rule defined on a key (a column or set of columns) in one table that guarantees that the values in that key match the values in a key in a related table (the referenced value).

Referential integrity also includes the rules that dictate what types of data manipulation are allowed on referenced values and how these actions affect dependent values.

See Also: key

roll back

Undo any changes to data that have been performed by SQL statements within an uncommitted transaction. After a transaction has been committed, it cannot be rolled back.

Oracle uses rollback segments to store old values. The redo log contains a record of changes.

See Also: commit, transaction, rollback segment

rollback segment

Logical database structure created by the database administrator to temporarily store undo information. Rollback segments store old data changed by SQL statements in a transaction until it is committed.

See Also: commit, logical structures, segment

row

Set of attributes or values pertaining to one entity or record in a table. A row is a collection of column information corresponding to a single record.

See Also: column, table

ROWID

A globally unique identifier for a row in a database. It is created at the time the row is inserted into a table, and destroyed when it is removed from a table.

schema

Collection of database objects, including logical structures such as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links.

A schema has the name of the user who controls it.

See Also: logical structures

SDTZ

Current session time zone.

segment

Third level of logical database storage. A segment is a set of extents, each of which has been allocated for a specific data structure, and all of which are stored in the same tablespace.

See Also: extent, data block

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.

See Also: client, client/server architecture

server processes

Server processes handle requests from connected user processes. A server process is in charge of communicating with the user process and interacting with Oracle to carry out requests of the associated user process.

See Also: process, user processes

session

Specific connection of a user to an Oracle instance through a user process. A session lasts from the time the user connects until the time the user disconnects or exits the database application.

See Also: connection, instance, user processes

shared pool

Portion of the system global area that contains shared memory constructs such as shared SQL areas. A shared SQL area is required to process every unique SQL statement submitted to a database.

See Also: system global area (SGA), SQL

shared server

A database server configuration that allows many user processes to share a small number of server processes, minimizing the number of server processes and maximizing the use of available system resources.

See Also: dedicated server

SQL

Structured Query Language, a nonprocedural language to access data. Users describe in SQL what they want done, and the SQL language compiler automatically generates a procedure to navigate the database and perform the desired task.

See Also: SQL*Plus, PL/SQL

SQL*Plus

Oracle tool used to execute SQL statements against an Oracle database. Oracle SQL includes many extensions to the ANSI/ISO standard SQL language.

See Also: SQL, PL/SQL

subtype

In the hierarchy of user-defined datatypes, a subtype is always a dependent on its supertype.

supertype

See: subtype

system global area (SGA)

A group of shared memory structures that contain data and control information for one Oracle database instance. If multiple users are concurrently connected to the same instance, then the data in the instance's SGA is shared among the users. Consequently, the SGA is sometimes referred to as the shared global area.

See Also: instance

table

Basic unit of data storage in an Oracle database. Table data is stored in rows and columns.

See Also: column, row

tablespace

A database storage unit that groups related logical structures together.

See Also: logical structures

temporary segment

Temporary segments are created by Oracle when a SQL statement needs a temporary work area to complete execution. When the statement finishes execution, the temporary segment's extents are returned to the system for future use.

See Also: extent, segment

transaction

Logical unit of work that contains one or more SQL statements. All statements in a transaction are committed or rolled back together.

See Also: commit, roll back

trigger

Stored database procedure automatically invoked whenever a table or view is modified, for example by `INSERT`, `UPDATE`, or `DELETE` operations.

Unicode

A way of representing all the characters in all the languages in the world. Characters are defined as a sequence of codepoints, a base codepoint followed by any number of surrogates. There are 64K codepoints.

Unicode column

A column of type `NCHAR`, `NVARCHAR2`, or `NCLOB` in Oracle9*i*, Release 1 (9.0.1), or later. It is guaranteed to be able to hold unicode.

UNIQUE key constraint

A data integrity constraint requiring that every value in a column or set of columns (key) be unique—that is, no two rows of a table have duplicate values in a specified column or set of columns.

See Also: integrity constraint, key

user name

The name by which a user is known to the Oracle server and to other users. Every user name is associated with a password, and both must be entered to connect to an Oracle database.

user processes

User processes execute the application or Oracle tool code.

See Also: process, Oracle processes

UTC

Coordinated Universal Time, previously called Greenwich Mean Time, or GMT.

view

A view is a custom-tailored presentation of the data in one or more tables. A view can also be thought of as a "stored query." Views do not actually contain or store data; they derive their data from the tables on which they are based.

Like tables, views can be queried, updated, inserted into, and deleted from, with some restrictions. All operations performed on a view affect the base tables of the view.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y