Oracle 8*i*

Concepts

Release 2 (8.1.6)

December 1999

Part No. A76965-01

ORACLE

Concepts, Release 2 (8.1.6)

Part No. A76965-01

Primary Authors: Lefty Leverenz, Diana Rehfield, Cathy Baird

Contributing Authors: Lance Ashdown, Steve Bobrowski, Cynthia Chin-Lee, Cindy Closkey, Bill Creekbaum, Jason Durbin, John Frazzini, Richard Mateosian, Denis Raphaely, John Russell, Danny Sokolsky, Randy Urbano

Contributors: Richard Allen, David Anderson, Andre Bakker, Mark Bauer, Ruth Baylis, Bill Bridge, Atif Chaudry, Jeff Cohen, Michele Cyran, Benoit Dageville, Mary Ann Davidson, Sandy Dreskin, Ahmed Ezzat, Jim Finnerty, Diana Lorentz, Anurag Gupta, Gary Hallmark, Michael Hartstein, Terry Hayes, Alex Ho, Chin Hong, Ken Jacobs, Sandeep Jain, Amit Jasuja, Hakan Jakobsson, Bob Jenkins, Ashok Joshi, Mohan Kamath, Jonathan Klein, R. Kleinro, Robert Kooi, Vishu Krishnamurthy, Muralidhar Krishnaprasad, Andre Kruglikov, Tirthankar Lahiri, Juan Loaiza, Brom Mahbod, William Maimone, Andrew Mendelsohn, Reza Monajjemi, Mark Moore, Rita Moran, Bhagat Nainani, Denise Oertel, Bruce Olsen, Robert Pang, Mark Porter, Maria Pratt, Tuomas Pystynen, Ann Rhee, Patrick Ritto, Hasan Rizvi, Sriram Samu, Hari Sankar, Gordon Smith, Mark Smith, Leng Leng Tan, Lynne Thieme, Alvin To, Alex Tsukerman, William Waddington, Joyo Wijaya, Linda Willis, Andrew Witkowski, Mohamed Zait

Graphic Designer: Valarie Moore

Contents

3 Tablespaces and Datafiles

4 Data Blocks, Extents, and Segments

Part III The Oracle Instance

5 Database and Instance Startup and Shutdown

6 Distributed Processing

7 Memory Architecture

8 Process Architecture

9 Database Resource Management

Part IV The Object-Relational DBMS

10 Schema Objects

11 Partitioned Tables and Indexes

12 Built-In Datatypes

13 User-Defined Datatypes

14 Object Views

Part V Data Access

15 SQL and PL/SQL

16 Transaction Management

17 Procedures and Packages

18 Advanced Queuing

19 Triggers

20 Oracle Dependency Management

Part VI Optimization of SQL Statements

21 The Optimizer

Part VII Parallel SQL and Direct-Load INSERT

22 Direct-Load INSERT

23 Parallel Execution of SQL Statements

Part VIII Data Protection

24 Data Concurrency and Consistency

25 Data Integrity

26 Controlling Database Access

27 Privileges, Roles, and Security Policies

28 Auditing

29 Database Recovery

Part IX Distributed Databases and Replication

30 Distributed Database Concepts

31 Replication

Send Us Your Comments

Oracle8*i* Concepts, Release 8.1.6

Part No. A76965-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- E-mail - infodev@us.oracle.com
- FAX - 1-650-506-7228. Attn: Oracle Server Documentation
- Postal service:
  Oracle Corporation
  Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

  If you would like a reply, please give your name, address, and telephone number below.

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

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

You can send comments to us in the following ways

- E-mail - infodev@us.oracle.com
- FAX -1-650-506-7228. Attn: Oracle Server Documentation
- Postal service
  Oracle Corporation
  Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

Please provide the following information:

Name:

Title:

Company:

Department:

E-Mail Address:

Postal Address:

Phone Number:

Book Title:

Part Number:

If you like, you can use the following questionnaire to give us feedback:

- 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 about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available).

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

Preface

This two-volume 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.

Oracle8*i* and Oracle8*i* Enterprise Edition

Oracle8i Concepts contains information that describes the functionality of the Oracle8*i* (also known as the standard edition) and the Oracle8*i* Enterprise Edition products. Oracle8*i* and Oracle8*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 and the Parallel Server option.

For information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8i*.

Audience

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

What You Should Already Know

You should be familiar with relational database concepts and with the operating system environment under which you are running Oracle. As a prerequisite, **all readers should read** Chapter 1, "Introduction to the Oracle Server". That chapter is a comprehensive introduction to the concepts and terminology used throughout the remainder of this manual.

If You Are Interested in Installation and Migration

This manual is not an installation or migration guide. Therefore, if your primary interest is installation, refer to your operating system-specific Oracle documentation, or if your primary interest is database and application migration, refer to *Oracle8i Migration*.

If You Are Interested in Database Administration

While this manual describes the architecture, processes, structures, and other concepts of the Oracle server, it does not explain how to administer the Oracle server. For that information, see *Oracle8i Administrator's Guide*.

If You Are Interested in Application Design

In addition to administrators, experienced users of Oracle and advanced database application designers will find information in this manual useful. However, database application developers should also refer to *Oracle8i Application Developer's Guide - Fundamentals* and to the documentation for the tool or language product they are using to develop Oracle database applications.

If You Are Interested in Optimizing Performance

If you are responsible for the efficient operation, maintenance, and performance of Oracle, you will find useful information in this manual. For more advanced information, refer also to *Oracle8i Designing and Tuning for Performance.*

How This Manual Is Organized

This manual is divided into two volumes, which contain the following parts:

- Volume 1

- Part I: What Is Oracle?
- Part II: Database Structures
- Part III: The Oracle Instance
- Part IV: The Object-Relational DBMS

- Volume 2
  - Part V: Data Access
  - Part VI: Optimization of SQL Statements
  - Part VII: Parallel SQL and Direct-Load INSERT
  - Part VIII: Data Protection
  - Part IX: Distributed Databases and Replication
  - Part X: Appendix

VOLUME 1

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.

Part II: Database Structures

Chapter 2, "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.

Chapter 3, "Tablespaces and Datafiles"

This chapter discusses how physical storage space in an Oracle database is divided into logical divisions called tablespaces. The physical operating system files associated with tablespaces, called datafiles, are also discussed.

Chapter 4, "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. The space

management background information here supplements that in the following chapter and in Chapter 10, "Schema Objects".

Part III: The Oracle Instance

Chapter 5, "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 6, "Distributed Processing"

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

Chapter 7, "Memory Architecture"

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

Chapter 8, "Process Architecture"

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

Chapter 9, "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 10, "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 11, "Partitioned Tables and Indexes"

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

Chapter 12, "Built-In 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 13, "User-Defined Datatypes"

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

Chapter 14, "Object Views"

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

VOLUME 2

Part V: Data Access

Chapter 15, "SQL and PL/SQL"

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.

Chapter 16, "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 17, "Procedures and Packages"

This chapter discusses the procedural language constructs called procedures, functions, and packages, which are PL/SQL program units that are stored in the database.

Chapter 18, "Advanced Queuing"

This chapter describes the Oracle Advanced Queuing feature, which allows users to store messages in queues for deferred retrieval and processing by the Oracle server.

Chapter 19, "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 20, "Oracle Dependency Management"

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

Part VI: Optimization of SQL Statements

Chapter 21, "The Optimizer"

This chapter introduces the optimizer, which is the part of Oracle that chooses the most efficient way to execute each SQL statement.

Part VII: Parallel SQL and Direct-Load INSERT

Chapter 22, "Direct-Load INSERT"

This chapter describes the direct-load insert path, which can be performed serially or in parallel, and the NOLOGGING clause.

Chapter 23, "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.

Part VIII: Data Protection

Chapter 24, "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 25, "Data Integrity"

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

Chapter 26, "Controlling Database Access"

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

Chapter 27, "Privileges, Roles, and Security Policies"

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

Chapter 28, "Auditing"

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

Chapter 29, "Database Recovery"

This chapter describes the files and structures used for database recovery and discusses how to protect an Oracle database from possible failures.

Part IX: Distributed Databases and Replication

Chapter 30, "Distributed Database Concepts"

This chapter discusses distributed database architecture, remote data access, and table replication.

Chapter 31, "Replication"

This chapter discusses replication of Oracle databases in a distributed database system.

Part X: Appendix

Appendix A, "Operating System-Specific Information"

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

How to Use This Manual

Every reader of this manual should read Chapter 1, "Introduction to the Oracle Server". This overview of the concepts and terminology related to Oracle provides a foundation for the more detailed information that follows in later chapters.

Each part of this manual addresses a specific audience within the general audiences previously described. For example, after reading Chapter 1, administrators who are interested primarily in managing security should focus on the information presented in Part VII, "Data Protection", particularly Chapter 26, "Controlling Database Access", Chapter 27, "Privileges, Roles, and Security Policies", and Chapter 28, "Auditing".

Conventions Used in This Manual

This manual uses different fonts to represent different types of information.

Text of the Manual

The text of this manual uses the following conventions.

UPPERCASE Characters

Uppercase text is used to call attention to command keywords, database object names, parameters, filenames, and so on.

For example, "After inserting the default value, Oracle checks the FOREIGN KEY integrity constraint defined on the DEPTNO column," or "If you create a private rollback segment, the name must be included in the ROLLBACK_SEGMENTS initialization parameter."

Italicized Characters

Italicized words within text are book titles or emphasized words.

Code Examples

Commands or statements of SQL, Oracle Enterprise Manager line mode (Server Manager), and SQL*Plus appear in a monospaced font.

For example:

```
INSERT INTO emp (empno, ename) VALUES (1000, 'SMITH');
ALTER TABLESPACE users ADD DATAFILE 'users2.ora' SIZE 50K;
```

Example statements may include punctuation, such as commas or quotation marks. All punctuation in example statements is required. All example statements terminate with a semicolon (;). Depending on the application, a semicolon or other terminator may or may not be required to end a statement.

UPPERCASE in Code Examples

Uppercase words in example statements indicate the keywords within Oracle SQL. When you issue statements, however, keywords are not case sensitive.

`lowercase` in Code Examples

Lowercase words in example statements indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file.

Your Comments Are Welcome

We value and appreciate your comment as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the following e-mail address:

infodev@us.oracle.com

If you prefer, you can send letters or faxes containing your comments to:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

Fax: (650) 506-7228

Part I

What Is Oracle?

Part I provides an overview of Oracle server concepts and terminology. It contains one chapter:

- Chapter 1, "Introduction to the Oracle Server"

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
- Distributed Processing and Distributed Databases
- Startup and Shutdown Operations
- Database Security
- Database Backup and Recovery
- Data Access

Note: This chapter contains information relating to both Oracle8*i* and the Oracle8*i* Enterprise Edition. Some of the features and options documented in this chapter are available only if you have purchased the Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for information about the differences between Oracle8*i* and the Oracle8*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 efficient and effective solutions with the following features:

client / server environments (distributed processing)	To take full advantage of a given computer system or network, Oracle allows 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 application concentrate on the interpretation and display of data.
large databases and space management	Oracle supports the largest of databases, which can contain terabytes of data. To make efficient use of expensive hardware devices, Oracle allows 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 allows 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 suffer from slow processing performance.
high availability	At some sites, Oracle works 24 hours per 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 sub-database level. For example, an administrator can disallow use of a specific application so that 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 allows administrators to manage heterogeneous systems with a single administration interface.
manageable security	To protect against unauthorized database access and use, Oracle provides fail-safe 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 non-distributed systems, yet receive the advantages of local database management. Oracle also offers the heterogeneous option that allows 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 open, comprehensive, and 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.

The Oracle Parallel Server: Multiple Instance Systems

Note: The Oracle Parallel Server option is available only if you have purchased the Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for details about the features and options available with Oracle8*i* Enterprise Edition.

Some hardware architectures (for example, shared disk systems) allow multiple computers to share access to data, software, or peripheral devices. Oracle with the

Parallel Server option can take advantage of such architecture by running multiple instances that share a single physical database. In appropriate applications, the Oracle Parallel Server allows access to a single database by the users on multiple machines with increased performance.

> **See Also:** *Oracle8i Parallel Server Concepts* for more information about the Oracle Parallel Server

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

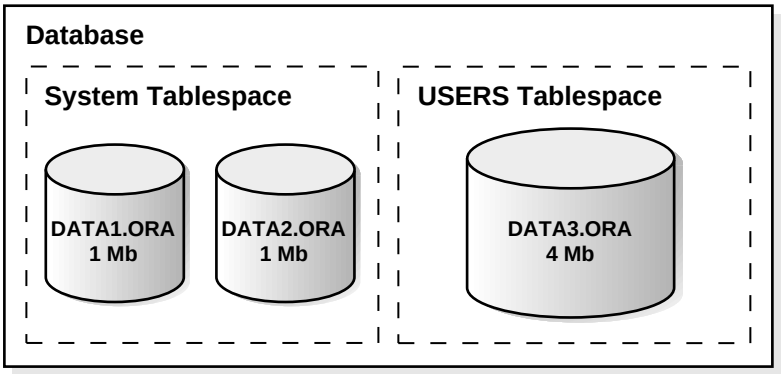
The logical structures of an Oracle database include tablespaces, schema objects, data blocks, extents, and segments.

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 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 may be 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: "Schemas and Schema Objects" on page 1-22 for more information about schema objects

Data Blocks, Extents, and Segments

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

See Also: Chapter 4, "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. A data block size is specified for each Oracle database when the database is created. A database uses and allocates free database space in Oracle data blocks.

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 non-clustered 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	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 rollback uncommitted transactions for users

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 1-30
- "Database Backup and Recovery" on page 1-44

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*. A database's 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 allow them to 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.

The Use of Datafiles The 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 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*), each of which is a group of change vectors describing a single atomic change to the database.

The primary function of the redo log is to record all changes made to data. Should a failure prevent modified data from being permanently written to the datafiles, the changes can be obtained from the redo log and 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 Use of Redo Log Files 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" on page 1-44 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 allows the control file to 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" on page 1-44
- Chapter 29, "Database Recovery"

Structured Query Language (SQL)

SQL (pronounced SEQUEL) is the programming language that defines and manipulates the database. SQL databases are relational databases; this means simply that data is stored in a set of simple relations. A database can have one or more tables. Each table has columns and rows. A table that has an employee database, for example, might have a column called employee number and each row in that column would be an employee's employee number.

You can define and manipulate data in a table with SQL statements. You use 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 data manipulation language (DML). DML statements include statements to alter and fetch data. The most

common SQL statement is the SELECT statement, which allows you to retrieve 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 allows you to control the flow of a SQL program, to use variables, and to 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 The Export utility provides a simple way for you to transfer 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 via 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 via 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 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 like offline instantiation.

> **See Also:** *Oracle8i Replication* for more information about Oracle Advanced Replication

SQL*Loader 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:** *Oracle8i 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. Among other things, the architectural features discussed in this section provide an understanding of the capabilities of the Oracle server to support:

- Many users concurrently accessing a single database
- The high performance required by concurrent multi-user, 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.

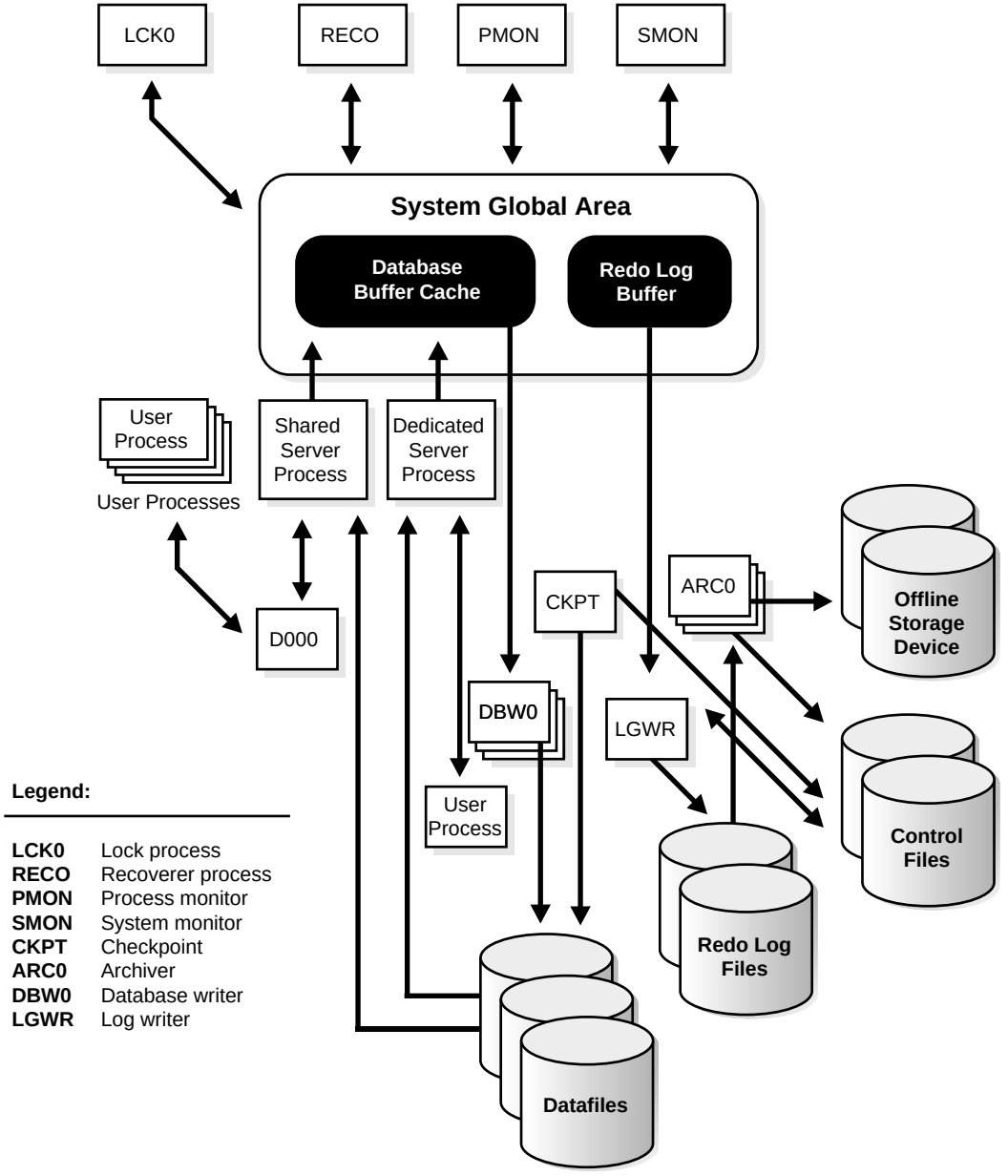
Processes are jobs or tasks that work in the memory of these computers.

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 *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 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. Its size is static.

Shared Pool 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 1-51 for more information about shared SQL areas.

Large Pool 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 multi-threaded server and Oracle XA.

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 the automatic cursor handling of the 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 specifically used for the parsing of SQL statements embedded within the application. The application developer can code an application so that it controls the phases of SQL statement execution and thus improve 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 configuration of Oracle.

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 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. They include server processes and background processes.

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 per server process. In a *dedicated server configuration*, a server process handles requests for a single user process. A *multi-threaded server configuration* allows many user processes to share a small number of server processes, minimizing the number of server processes and maximizing the utilization 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 multi-threaded 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 may use several background processes. The names of these processes are DBW*n*, LGWR, CKPT, SMON, PMON, ARC*n*, RECO, D*nnn*, LCK0, SNP*n*, 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.

Since 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 1-52 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 system global area (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 signalling 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 Oracle Parallel Server), 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 (see below) and server processes and restarts them if they have failed.

Archiver (ARC*n*) The *archiver* copies the online redo log files to archival storage when they are full or a log switch occurs. Although a single ARC*n* process (ARC0) is sufficient for most systems, you can specify up to ten 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 ten 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 1-46 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.

Dispatcher (D*nnn*) *Dispatchers* are optional background processes, present only when a multi-threaded 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 (LCK0) The *lock* process (LCK0) is used for inter-instance locking in the Oracle Parallel Server.

> **See Also:** "The Oracle Parallel Server: Multiple Instance Systems" on page 1-4 for more information about the configuration of the lock process.

Job Queue (SNP*n*) In a distributed database configuration, up to thirty-six *job queue processes* (SNP0, ..., SNP9, SNPA, ..., SNPZ) can automatically refresh table snapshots. These processes wake up periodically and refresh any snapshots that are scheduled to be automatically refreshed. If more than one job queue process is used, the processes share the task of refreshing snapshots. These processes also execute job requests created by the DBMS_JOB package and propagate queued messages to queues on other databases.

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

Communications Software and Net8

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 Net8. *Dispatchers* are optional background processes, present only in the multi-threaded server configuration. *Net8* is Oracle's interface to standard communications protocols that allows for the proper transmission of data between computers.

See Also: "Oracle and Net8" on page 1-36

The Program Interface

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 via 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 Net8 driver.
3. The server is running the proper Net8 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 an identical 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 that 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 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 appropriate 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 only the most basic level of operations that Oracle performs.

> **See Also:** Chapter 8, "Process Architecture" for more information about Oracle configuration.

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 aspects:

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 allow 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 allows 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.

Object types are abstractions of the real-world entities—for example, purchase orders—that application programs deal with. 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 like C 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.

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.)

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 may 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 11, "Partitioned Tables and Indexes" for more information about partitioning.
- "Data Integrity" on page 1-57 for more information about business rules.

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 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.
- Simplify statements for the user. For example, views allow users to select information from multiple tables without requiring the users to actually know how to perform a correlated subquery.
- 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 might 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 10-16 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 master tables can improve query performance through *query rewrites*. For queries that involve aggregate data or joins, the optimizer can rewrite the query to access the precomputed results stored in a materialized view. Query rewrites are particularly useful in a data warehouse environment.

Another name for materialized view is *snapshot*. This term generally refers to a materialized view used for replicating data in a remote database. In SQL statements, the keywords SNAPSHOT and MATERIALIZED VIEW are synonymous.

> **See Also:** "Table Replication" on page 1-36 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 input 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" on page 1-50 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, system-wide use by any database user.

Indexes

Indexes are optional structures associated with tables, which 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 may be slower.

You can *partition* indexes.

> **See Also:** Chapter 11, "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.

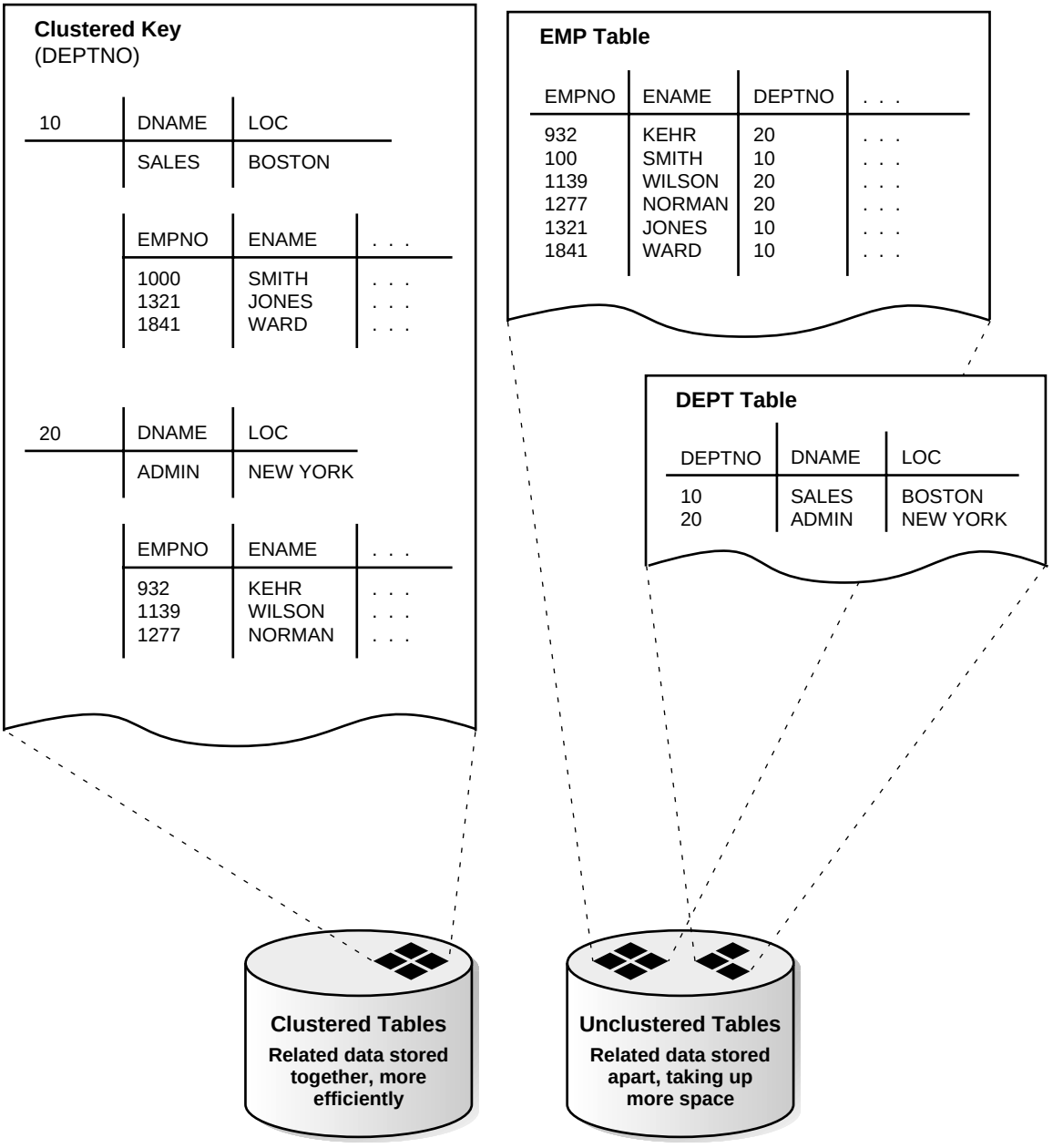
The related columns of the tables in a cluster are called the *cluster key*. The cluster key is indexed so that rows of the cluster can be retrieved with a minimum amount of I/O. Because the data in a cluster key of an index cluster (a non-hash cluster) is stored only once for multiple tables, clusters may 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 via SQL in the same way as data stored in a non-clustered table.

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

Figure 1–3 Clustered and Non-clustered Tables



Hash Clusters *Hash clusters* also 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 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, you connect the tables by inner equijoins using the JOIN clause of the HIERARCHY clause.

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 1-34
- Chapter 30, "Distributed Database Concepts"

The 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 2, "The Data Dictionary"

Data Concurrency and Consistency

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 multi-user 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*, or 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 his or her 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 1-54 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, Rollback Segments, 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 rollback segments. 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 rollback segments 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—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 doing no 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 allowing 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 sophisticated 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 obtained on a resource (such as a row or a table); however, many share locks can be obtained 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.

Distributed Processing and Distributed Databases

As computer networking becomes more and 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.

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* portion and a back-end or a *server* portion.

The Client

The client portion is the front-end database application and 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. The client workstation can be optimized for its job. For example, it might not need large disk capacity or it might benefit from graphic capabilities.

The Server

The server portion runs Oracle software and handles the functions required for concurrent, shared data access. The server portion receives and processes the SQL and PL/SQL statements that originate from client applications. The computer that manages the server portion can be optimized for its duties. For example, it can have large disk capacity and fast processors.

Multi-Tier Architecture: Application Servers

A *multi-tier 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* is a process that 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 serves as the repository for most of the data used in the operation

This architecture allows you to use 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 (client/server architecture, discussed previously).

While a distributed database allows increased access to a large amount of data across a network, it must also provide the ability to hide the location of the data and the complexity of accessing it across the network. The distributed DBMS 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, a reference is made including 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, you 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 might occur, 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.

Complete Transparency to Database Users The Oracle two-phase commit mechanism is completely transparent to users that issue distributed transactions. A simple 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.

Automatic Recovery from System or Network Failures The RECO (recoverer) 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 of each local Oracle server automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

Optional Manual Override Capability 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 allows the local database administrator to free up any locked resources that may be held indefinitely as a result of the long-term failure.

Facilities for Distributed Recovery If a database must be recovered to a point in the past, Oracle's recovery facilities allow database administrators at other sites to return their databases to the earlier point in time also. This ensures that the global database remains consistent.

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 (snapshots).

Oracle and Net8

Net8 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 software called Net8.

Net8 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 Net8, 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: *Net8 Administrator's Guide*

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. Dismount 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 5-5 for more information about starting a database
- "Parameter Files" on page 5-4 for a sample initialization parameter file
- *Oracle8i Reference* for more information about initialization parameters
- "Database and Instance Shutdown" on page 5-10 for more information about shutting down an instance and the database to which it is connected

Database Security

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 distinct 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

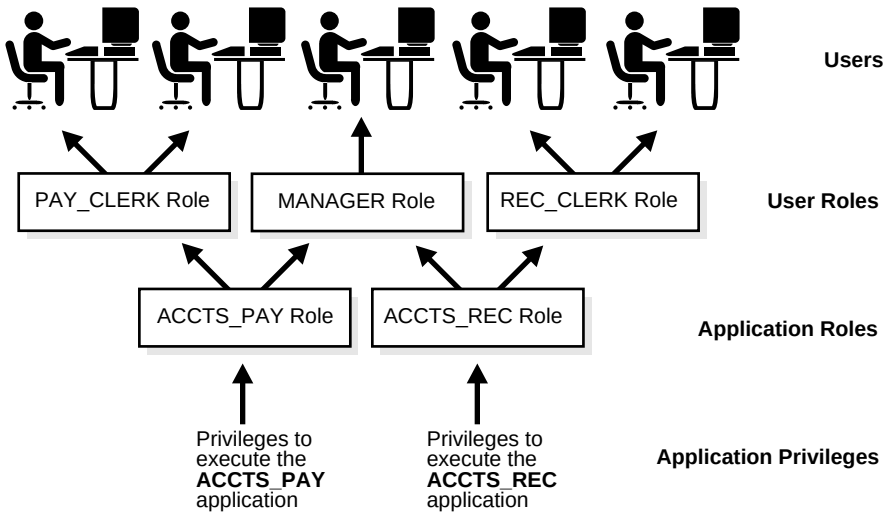
The Oracle server provides *discretionary access control*, which is a means of restricting access to information based on privileges. The appropriate privilege must be assigned to a user in order for that user to access a schema object. Appropriately privileged users can grant other users privileges 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 1–4 illustrates the relationships of the different Oracle security facilities, and the following sections provide an overview of users, privileges, and roles.

Figure 1–4 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 distinct 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 end-users so that they can use a database application to accomplish specific tasks.

Granting 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 allow for 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 are granted to users or other roles.

The following properties of roles allow for easier privilege management:

reduced granting of privileges	Rather than explicitly granting the same set of privileges to many users, a database administrator can grant the privileges for a group of related users to a role. The database administrator can then grant the role to each member of the group.
dynamic privilege management	When 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	The roles granted to a user can be selectively enabled (available for use) or disabled (not available for use). This allows specific control of a user's privileges in any given situation.
application awareness	A database application can be designed to enable and disable selective roles automatically when a user attempts to use the application.

Database administrators often create roles for a database application. The DBA grants an application role all privileges necessary to run the application. The DBA then grants the 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.

Storage Settings and Quotas

Oracle provides means for directing and limiting the use of disk space allocated to the database on a per user basis, 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:

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

Auditing

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 allow 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 the use 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 accesses 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 allows 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 referred to as the *audit trail*. Predefined views of the audit trail are available so that you can easily retrieve audit records.

Database Backup and Recovery

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 that users of the system can continue to work

Why Is Recovery Important?

In every database system, the possibility of a system or hardware failure always exists. Should a failure occur and affect 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 below:

user error	User errors can require a database to be recovered to a point in time before the error occurred. For example, a user might accidentally drop a table. To allow 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 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 may 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 Oracle Parallel Server 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 arise 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 may 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 that 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 and quick 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, rollback segments, a control file, and database backups.

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; 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.

LogMiner SQL-Based Log Analyzer LogMiner is a relational tool that allows you to read, analyze, and interpret online and archived log files using SQL. 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 allows you to 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:

- *Oracle8i Administrator's Guide* for more information about the LogMiner
- "Redo Log Files" on page 1-9 for more information about redo log files

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.

Rollback Segments

Rollback segments record rollback 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.

See Also: "Data Blocks, Extents, and Segments" on page 1-7 for more information about rollback segments

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

Due to the way in which DBW*n* writes database buffers to datafiles, at any given time a datafile may contain some tentative modifications by uncommitted transactions and may 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 may only appear in the redo log. Therefore, the redo log contains committed data that must be applied to the datafiles.
- Since the redo log may have contained 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*, that is, 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 rollback segments 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.

Recovery Manager

Recovery Manager (RMAN) is an Oracle utility that manages backup and recovery operations, creating backups of database files (datafiles, control files, and archived redo log files) and restoring or recovering 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:** *Oracle8i Backup and Recovery Guide* for more information about Recovery Manager

Data Access

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. The SQL implemented by Oracle Corporation for Oracle is 100 percent compliant with the ANSI/ISO standard SQL data language.

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 execute:

```
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) *DDL 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" on page 1-37 for more information about privileges

Data Manipulation Language (DML) *DML 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 allow the user or application developer to group changes into logical transactions. Examples include COMMIT, ROLLBACK, and SAVEPOINT.

> **See Also:** "Transactions" on page 1-52 for more information about transaction control statements

Session Control Statements *Session control* statements allow a user to 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 allows you to 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

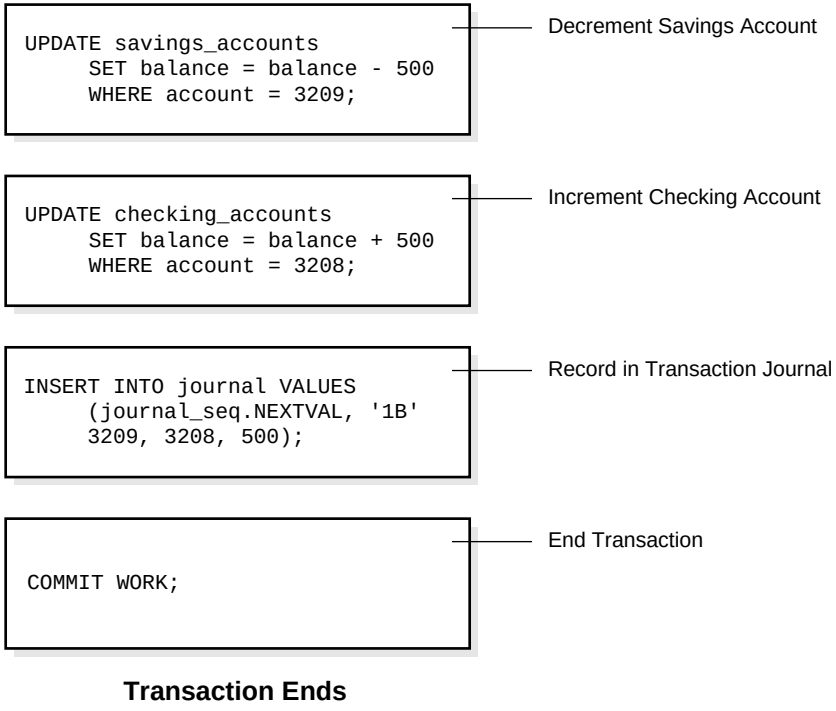
statement. A transaction ends when it is explicitly committed or rolled back (both terms are discussed later in this section) by that user.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might 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 either of the updates, then neither update is made.

Figure 1–5 illustrates the banking transaction example.

Figure 1–5 A Banking Transaction



Committing and Rolling 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 allows 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 provide the database user or application developer with the capability of guaranteeing 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 non-related 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

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, the users of PL/SQL 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 executed 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

Packages provide a method of encapsulating and storing related procedures, functions, variables, and other package constructs together as a unit in the database. While packages allow the administrator or application developer the ability 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) and performance (for example, all objects of the package are parsed, compiled, and loaded into memory once).

See Also:

- *Oracle8i Supplied PL/SQL Packages Reference*
- *Oracle8i Supplied Java Packages Reference*

Database Triggers

Oracle allows you to write 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. 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 may 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 as solutions 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:

- 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 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 may be defined per 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

Database triggers allow you to 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.

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 2, "The Data Dictionary"
- Chapter 3, "Tablespaces and Datafiles"
- Chapter 4, "Data Blocks, Extents, and Segments"

Additional Information: The following chapters describe other logical database structures:

- Chapter 10, "Schema Objects"
- Chapter 11, "Partitioned Tables and Indexes"

2

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

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 its associated 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, you 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 3-6 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. Therefore, 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 should keep strict control of this central account.

WARNING: 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.

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 that 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.

Caching of the Data Dictionary for Fast Access

Much of the data dictionary information is cached in the SGA (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.

Adding New Data Dictionary Items

You can add new tables or views to the data dictionary. If you add new data dictionary objects, the owner of the new objects should be the user SYSTEM or a third Oracle user.

> **Caution:** Never create new objects belonging to user SYS, except by running the script provided by Oracle Corporation for creating data dictionary objects.

Deleting Data Dictionary Items

All changes to the data dictionary are performed by Oracle in response to DDL statements, therefore **no data in any data dictionary tables should be deleted or altered by any user.**

The single exception to this rule is the table SYS.AUD$. When auditing is enabled, this table can grow without bound. Although you should not drop the AUDIT_TRAIL table, the security administrator can safely delete data from it because the rows are for information only and are not necessary for Oracle to run.

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 via the SQL language. Some views are accessible to all Oracle users; 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 2–1 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:** *Oracle8i 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 via 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

```
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:** *Oracle8i 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:** *Oracle8i Reference* for a complete list of the dynamic performance views' synonyms and their columns

3

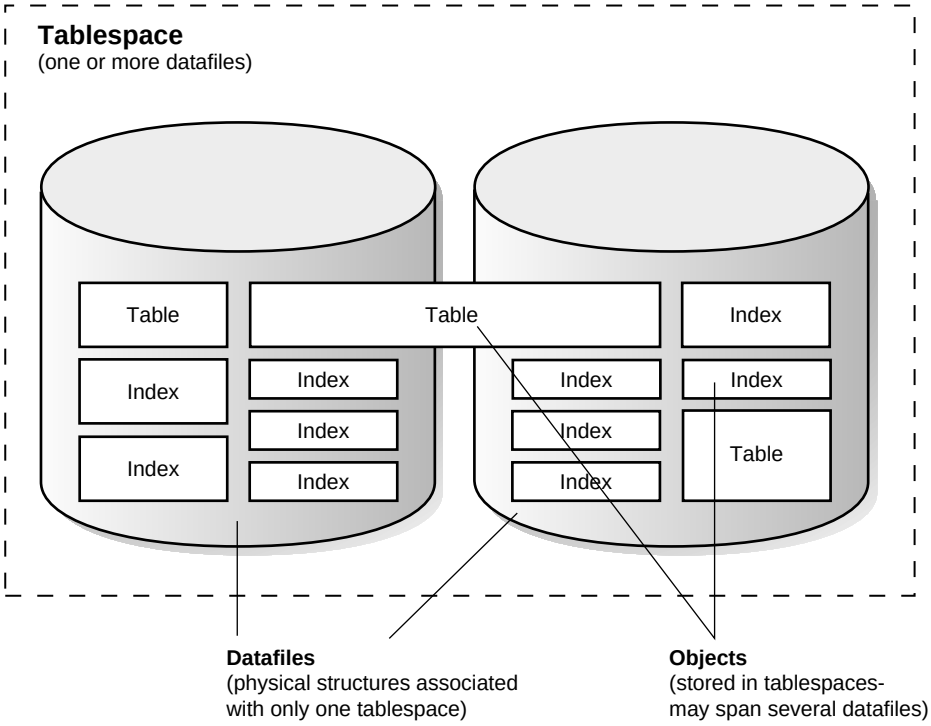
Tablespaces and Datafiles

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 Databases, Tablespaces, and Datafiles

Oracle stores data logically in *tablespaces* and physically in *datafiles* associated with the corresponding tablespace. Figure 3–1 illustrates this relationship.

Figure 3–1 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 might have three tablespaces, each consisting of two datafiles (for a total of six datafiles).

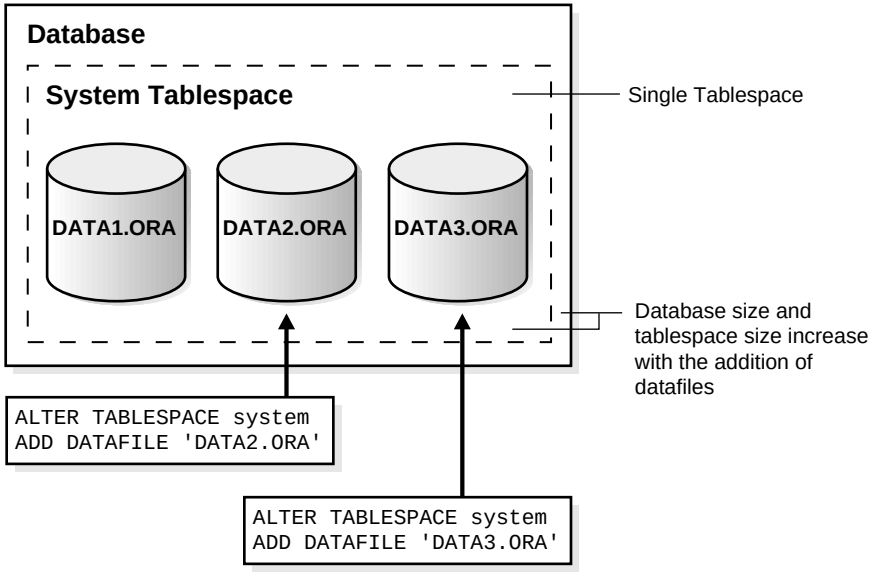
Allocating More Space for a Database

You can enlarge a database in three ways:

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

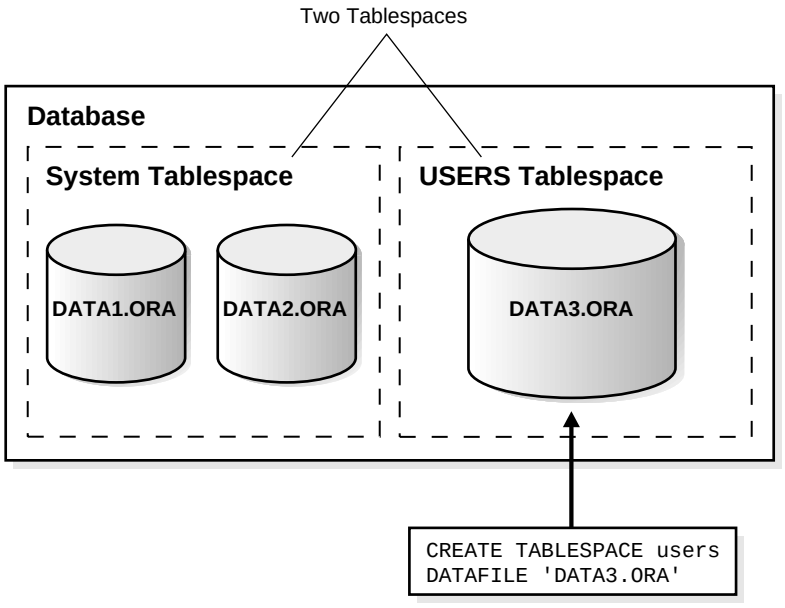
When you add another datafile to an existing tablespace, you increase the amount of disk space allocated for the corresponding tablespace. Figure 3–2 illustrates this kind of space increase.

Figure 3–2 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 3–3 illustrates this.

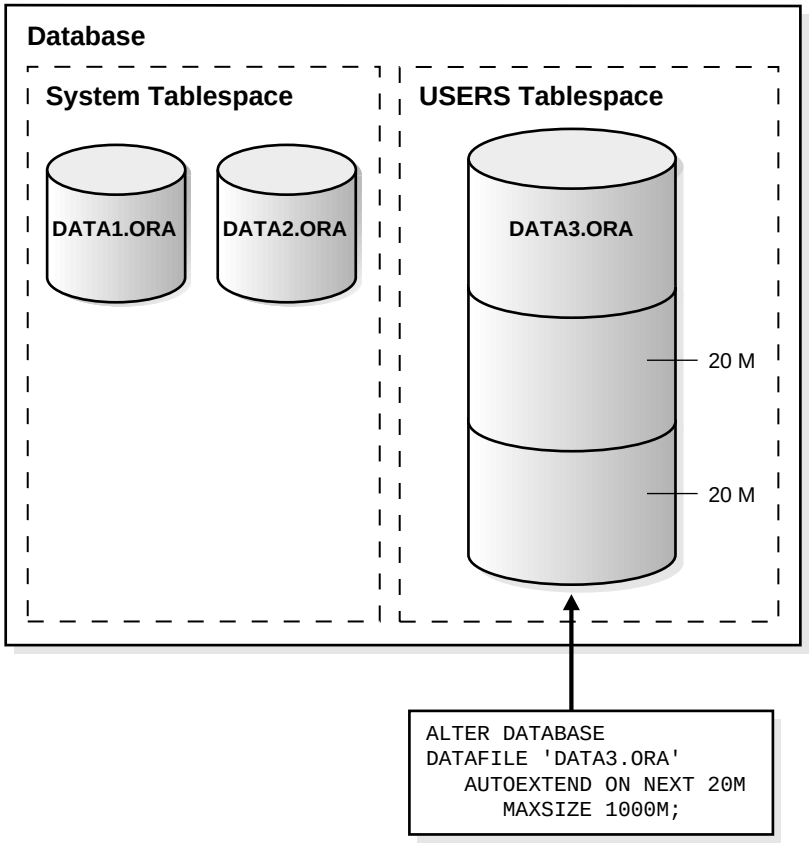
Figure 3–3 Enlarging a Database by Adding a New Tablespace



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 allow datafiles in existing tablespaces to grow dynamically as more space is needed. You accomplish this by altering existing files or by adding files with dynamic extension properties. Figure 3–4 illustrates this.

Figure 3–4 Enlarging a Database by Dynamically Sizing Datafiles



See Also: *Oracle8i Administrator's Guide* for more information about increasing the amount of space in your database

Tablespaces

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*.

This section includes the following topics about tablespaces:

- The SYSTEM Tablespace
- Using Multiple Tablespaces
- Space Management in Tablespaces
- Online and Offline Tablespaces
- Read-Only Tablespaces
- Temporary Tablespaces
- Transporting Tablespaces between Databases

> **See Also:** Chapter 4, "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

All data stored on behalf of stored PL/SQL program units (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 allow for the space they use in the SYSTEM tablespace.

See Also:

- "Online and Offline Tablespaces" on page 3-9 for more information about the permanent online condition of the SYSTEM tablespace
- Chapter 17, "Procedures and Packages" and Chapter 19, "Triggers" for more information about the space requirements of PL/SQL program units

Using Multiple Tablespaces

A small database might 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 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, and 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.

See Also: "Extents" on page 4-12

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.

This is the default method of space management in a tablespace. It was the only method available in Oracle releases 8.0 and earlier.

See Also: "Rollback Segments" on page 4-21 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.

The LOCAL clause of the EXTENT MANAGEMENT clause specifies this method of space management in various CREATE statements:

- For a permanent tablespace, you can specify EXTENT MANAGEMENT LOCAL in the CREATE TABLESPACE statement.

- For a temporary tablespace, you can specify EXTENT MANAGEMENT LOCAL in the CREATE TEMPORARY TABLESPACE statement.
- Currently the functionality to create a SYSTEM tablespace as locally managed is not supported. Please refer to bug 809225.

See Also:

- *Oracle8i SQL Reference* for details about SQL statements
- "Extents Managed Locally" on page 4-13 for more information about extent size
- "Temporary Tablespaces" on page 3-12 for more information about temporary tablespaces

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 might take a tablespace offline:

- To make a portion of the database unavailable, while allowing normal access to the remainder of the database
- To perform an offline tablespace backup (although a tablespace can be backed up while online and in use)
- To make an application and its group of tables temporarily unavailable while updating or maintaining the application

You cannot take a tablespace offline if it contains any rollback segments that are in use.

See Also: "Rollback Segments" on page 4-21 for more information about rollback segments

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 was 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 transferred from database to database.

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" on page 3-12 for more information about transferring online tablespaces between databases
- *Oracle8i Utilities* for more information about tools for data transfer

Using 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 will do 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.

Whenever you create a new tablespace, it is always created as read-write. You can change the tablespace to read-only with the READ ONLY clause of the ALTER TABLESPACE statement, making all of the tablespace's associated datafiles read-only as well.

The ALTER TABLESPACE ... READ ONLY statement places the tablespace in a *transitional read-only* mode and waits for existing transactions to complete (commit or roll back). This transitional state does not allow any further write operations to the tablespace except for the rollback of existing transactions that previously modified blocks in the tablespace. Hence, in transition the tablespace behaves like a read-only tablespace for all user statements except ROLLBACK. After all of the existing transactions have either committed or rolled back, the ALTER TABLESPACE ... READ ONLY statement completes and the tablespace is placed in read-only mode.

> **Note:** The transitional read-only state only occurs if the value of the initialization parameter COMPATIBLE is 8.1.0 or greater. For parameter values less than 8.1.0, the ALTER TABLESPACE ... READ ONLY statement will fail if any active transactions exist.

You can use the READ WRITE clause of the ALTER TABLESPACE statement to make a read-only tablespace read-write again.

Making a tablespace read-only does not change its offline or online status. Offline datafiles cannot be accessed. Bringing a datafile in a read-only tablespace online

makes the file only readable. The file cannot be written to unless its associated tablespace is returned to the read-write state. You can take the files of a read-only tablespace online or offline independently using the DATAFILE clause of the ALTER DATABASE statement.

Read-only tablespaces cannot be modified. To update a read-only tablespace, you must 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, should you need to recover your database, you do not need to recover any read-only tablespaces, because they could not have been modified. However, read-only tablespaces may need attention during instance or media recovery, depending upon whether and when they have ever been read-write.

You can drop items, such as tables and indexes, from a read-only tablespace, just as you can drop items from an offline tablespace. However, you cannot create or alter objects in a read-only tablespace.

You cannot add datafiles to a read-only tablespace, even if you take the tablespace offline. When you add a datafile, Oracle must update the file header, and this write operation is not allowed in a read-only tablespace.

See Also:

- *Oracle8i Administrator's Guide* for more information about changing a tablespace to read-only or read-write mode
- *Oracle8i SQL Reference* for more information about the ALTER TABLESPACE statement
- *Oracle8i Backup and Recovery Guide* for more information about recovery

Temporary Tablespaces

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 in Oracle Parallel Server environments.

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 4, "Data Blocks, Extents, and Segments" for more information about segments

Creating and Altering Temporary Tablespaces

You create temporary tablespaces by using the CREATE TABLESPACE or CREATE TEMPORARY TABLESPACE statement:

- For a dictionary-managed temporary tablespace, use the TEMPORARY clause of CREATE TABLESPACE.
- For a locally managed temporary tablespace, use CREATE TEMPORARY TABLESPACE. This statement specifies TEMPFILES instead of DATAFILES.

You can also change a tablespace from PERMANENT to TEMPORARY or vice versa by using the ALTER TABLESPACE statement for any temporary tablespace (either locally managed or dictionary-managed).

See Also:

- "Temporary Datafiles" on page 3-18 for information about TEMPFILES
- "Space Management in Tablespaces" on page 3-7 for information about locally managed and dictionary-managed tablespaces
- *Oracle8i SQL Reference* for information about the CREATE TABLESPACE, CREATE TEMPORARY TABLESPACE, and ALTER TABLESPACE statements
- *Oracle8i Designing and Tuning for Performance* for information about setting up temporary tablespaces for sorts and hash joins

Transporting Tablespaces between Databases

The *transportable tablespace* feature enables you to move a subset of an Oracle database from one Oracle database to another. 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.

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 that you do not have to rebuild the indexes after importing or loading the table data.

In the current release, you can transport tablespaces only between Oracle databases that use the same data block size and character set, and that run on compatible platforms from the same hardware vendor.

Moving or Copying 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:** *Oracle8i Administrator's Guide* for details about how to move or copy tablespaces to another database

Transportable Data Sets You can transport a data set consisting of one or more tablespaces, as long as the set of schema objects in the data set is self-contained. If you transport a data set that contains a pointer to a BFILE, you must also move the BFILE and set the directory correctly in the target database.

If the data set includes a partitioned table, it must contain all of the table's partitions. To transport a subset of a partitioned table, you can exchange the partitions into tables before transporting them.

> **See Also:** "REFs" on page 13-9 for information about moving object references

Tablespace Metadata The metadata that you export can include or omit information about triggers, grants, and constraints, depending on which export options you use. Primary key constraints are always exported.

Benefits of Transporting Tablespaces

Transporting tablespaces is particularly useful for:

- Data warehouses
- Data marts
- Data publication

You can also transport tablespaces to move or copy data between Oracle databases that have different compatibility or release levels.

> **See Also:** *Oracle8i Migration* for details about how to move or copy tablespaces between Oracle releases or compatibility levels

Data Warehouses and Data Marts An enterprise *data warehouse* contains historical detailed data about the company. Typically, data flows from one or more online transaction processing (OLTP) databases into the data warehouse on a monthly, weekly, or daily basis. The data is usually processed in a *staging database* before being added to the data warehouse.

A *data mart* contains a subset of corporate data that is of value to a specific business unit, department, or set of users. Typically, a data mart is derived from an enterprise data warehouse.

Transporting tablespaces can be useful for many purposes in a data warehouse environment:

- To move data from an OLTP database to a staging database, where the data can be cleaned and transformed before going into the data warehouse
- To move data from the staging database to the enterprise data warehouse. The new data can become a partition of the historical data by exchanging tables with partitions.
- To move data from a data warehouse to a data mart
- To archive obsolete data from the data warehouse, keeping the archived data together with its metadata so that the tablespace could be restored if necessary

> **See Also:** *Oracle8i Data Warehousing Guide* for more information about data warehouses and data marts

Data Publication *Content providers* acquire data and make it available in a useful format. For example, a content provider might acquire statistical data from hospitals and provide it to insurance companies, or a telephone company might give large customers their billing data on CDs. Content providers can transport tablespaces to publish structured data on CD or other media, enabling customers to integrate the published data into their Oracle databases.

Datafiles

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 might take a significant amount of time.he 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 4, "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 per tablespace and can simplify administration of datafiles.

> **See Also:** *Oracle8i Administrator's Guide* for more information about resizing datafiles

Offline Datafiles

You can take tablespaces *offline* (make unavailable) or bring them *online* (make available) 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.
- Media recovery does not recognize tempfiles:
  - BACKUP CONTROLFILE does not generate any information for tempfiles.
  - CREATE CONTROLFILE cannot specify any information about tempfiles.
- 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 3-7 for more information about locally managed tablespaces

4

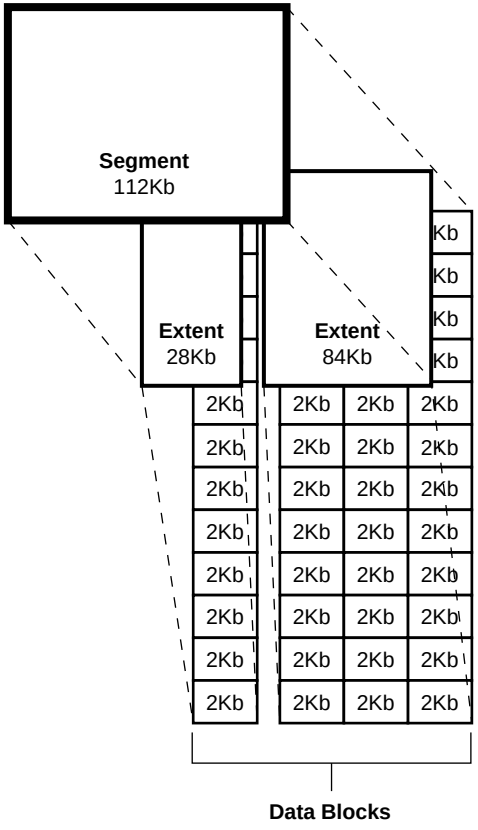
Data Blocks, Extents, and Segments

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

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

Figure 4–1 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.

Data Blocks

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 what is called a *block size*. Oracle requests data in multiples of Oracle data blocks, not operating system blocks.

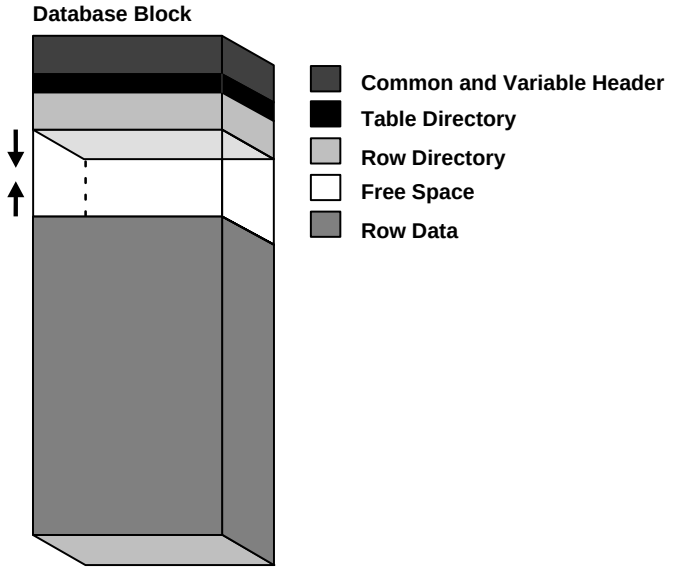
You set the data block size for each Oracle database when you create the database. This data block size should be a multiple of the operating system's block size within the maximum limit to avoid unnecessary I/O. Oracle data blocks are the smallest units of storage that Oracle can use or allocate.

> **See Also:** Your Oracle operating system-specific documentation for more information about data block sizes.

Data Block Format

The Oracle data block format is similar regardless of whether the data block contains table, index, or clustered data. Figure 4–2 illustrates the format of a data block.

Figure 4–2 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 tables 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 4-10

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 non-null 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: "An Introduction to PCTFREE, PCTUSED, and Row Chaining" on page 4-5 for more information about space management parameters.

An Introduction to PCTFREE, PCTUSED, and Row Chaining

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. You specify these parameters when creating or altering 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 12-12 for more information.

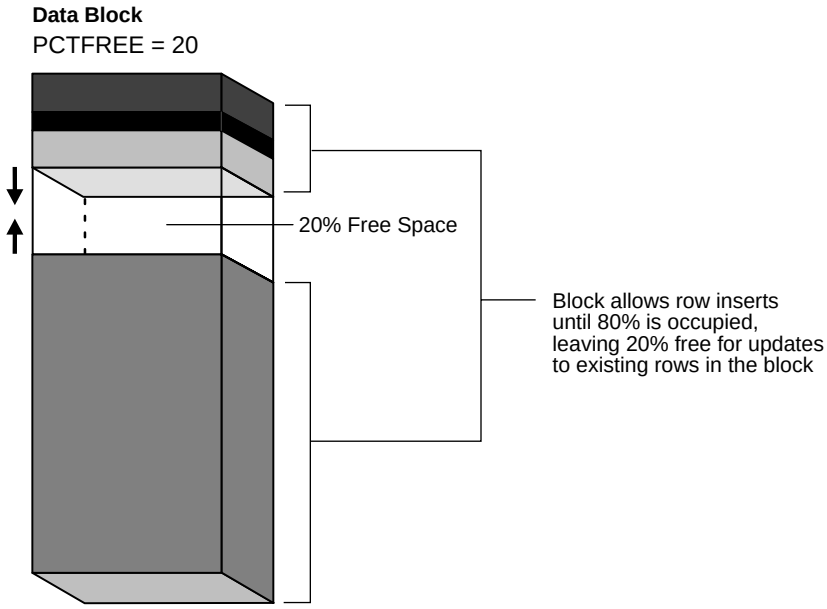
The PCTFREE Parameter

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 4–3 illustrates PCTFREE.

Figure 4–3 PCTFREE



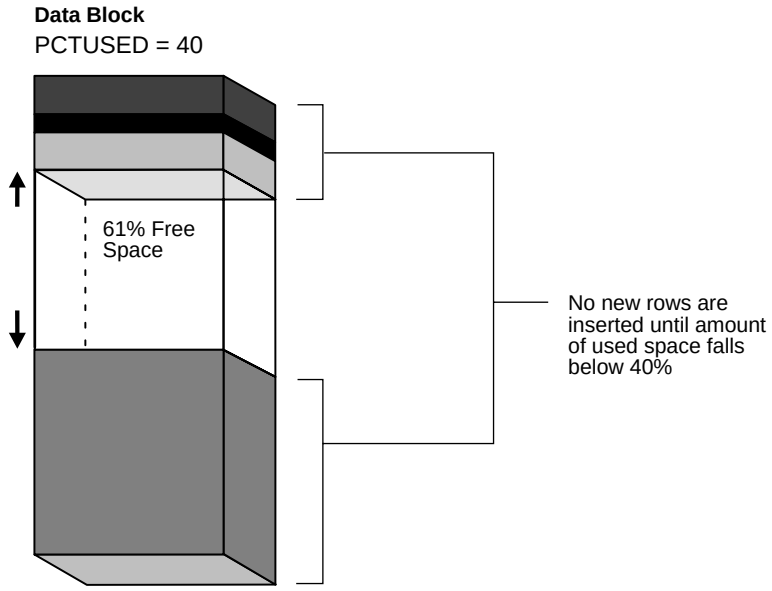
The PCTUSED Parameter

The PCTUSED parameter sets the minimum percentage of a block that can be *used* for row data plus overhead before new rows will be 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 4–4 illustrates this.

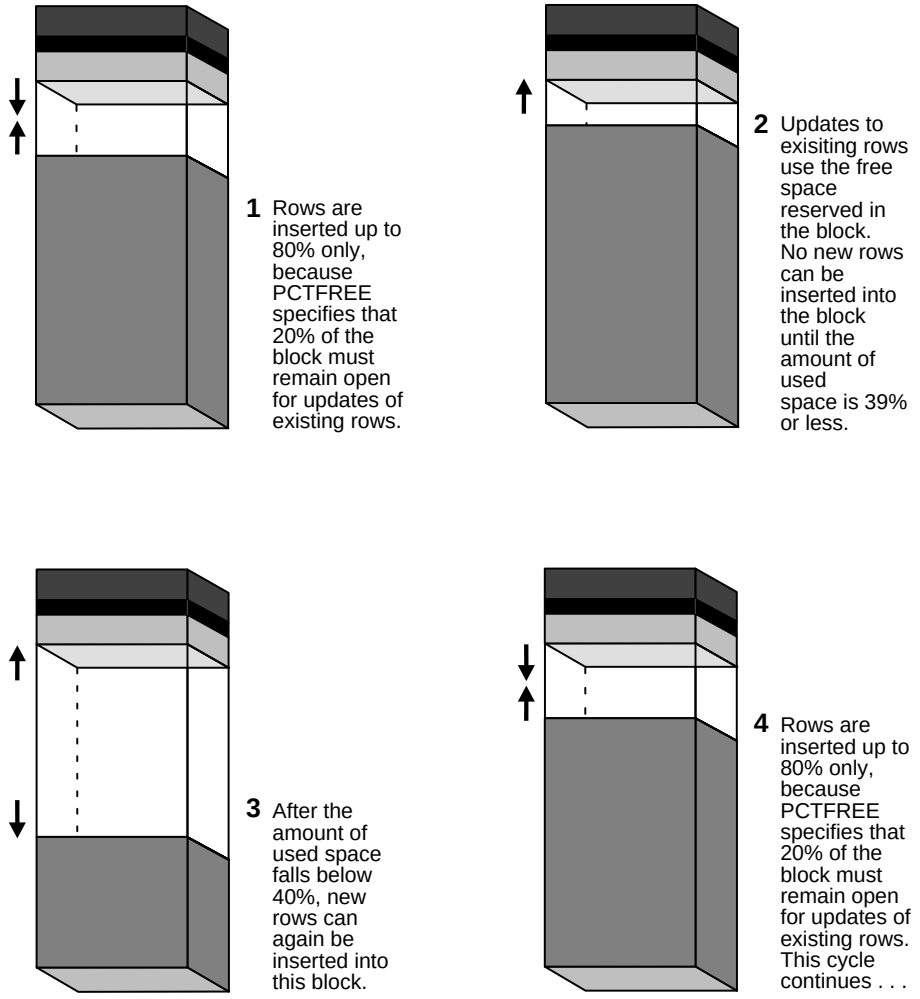
Figure 4–4 PCTUSED



How PCTFREE and PCTUSED Work Together

PCTFREE and PCTUSED work together to optimize the utilization of space in the data blocks of the extents within a data segment. Figure 4–5 illustrates the interaction of these two parameters.

Figure 4–5 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 per 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.

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 that 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 would decrease 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 10-5 for more information on the format of a row and a row piece
- "Rowids of Row Pieces" on page 10-8 for more information on rowids
- "Physical Rowids" on page 12-16 for information about rowids
- *Oracle8i Designing and Tuning for Performance* for information about reducing chained and migrated rows and improving I/O performance

Extents

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. (The next section explains the factors controlling the size of incremental extents.)

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 4-26
- "Free Space and Parallel DDL" on page 23-34

Determining 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 3-7

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:

- *Oracle8i Administrator's Guide*
- *Oracle8i 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 may deallocate 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:

- *Oracle8i Administrator's Guide*
- *Oracle8i 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 (which are known as snapshots and snapshot logs in a replication environment) in the same manner as for tables and clusters.

See Also: "Materialized Views" on page 10-18 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 to see if the rollback segments of the database 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 4-26

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:

- "Temporary Segments" on page 4-19
- "Temporary Tables" on page 10-11

Segments

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; 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:

- Data Segments
- Index Segments
- Temporary Segments
- Rollback Segments

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 snapshots and snapshot logs in the same manner as for tables and clusters.

See Also:

- *Oracle8i Replication* for information on snapshots and snapshot logs
- *Oracle8i SQL Reference* for information on the CREATE and ALTER statements

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.

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 Requiring Temporary Segments

The following statements may 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 may 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 list above, the database administrator may want to improve performance by adjusting the initialization parameter SORT_AREA_SIZE.

> **See Also:** *Oracle8i 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 10-11

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. You 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 may avoid fragmentation of the SYSTEM and other tablespaces that otherwise would 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 26, "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 10-11

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.

See Also:

- "Multiversion Concurrency Control" on page 24-4 for information about read consistency
- "Rolling Back Transactions" on page 16-6
- "Rollback Segments and Rolling Back" on page 29-9 for information about database recovery

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.)

Logging Rollback Entries

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 on a per-transaction basis. 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 allows the application developer or user to 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) may require a new extent to be allocated for the rollback segment.

See Figure 4–6 on page 4-24, Figure 4–7 on page 4-25, and Figure 4–8 on page 4-26 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.

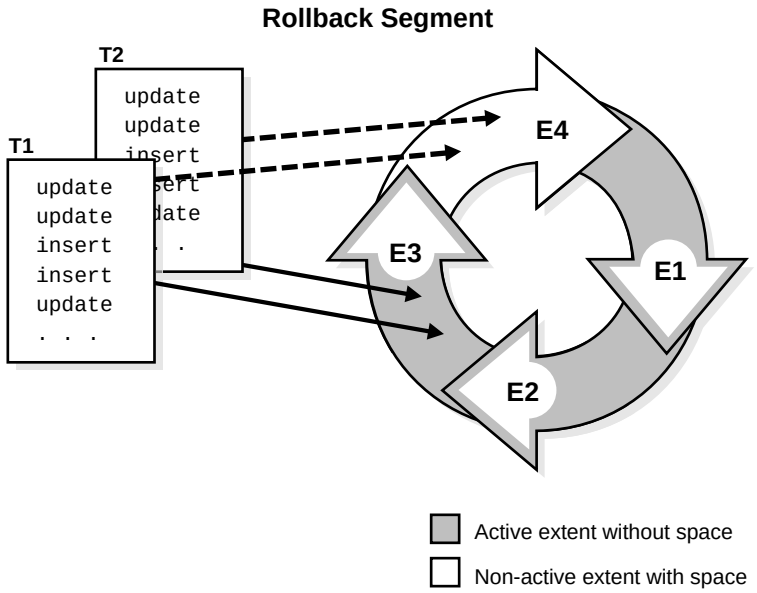
When a transaction runs out of space in the current extent and needs to continue writing, Oracle finds an available extent of the same rollback segment in one of two ways:

- 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.

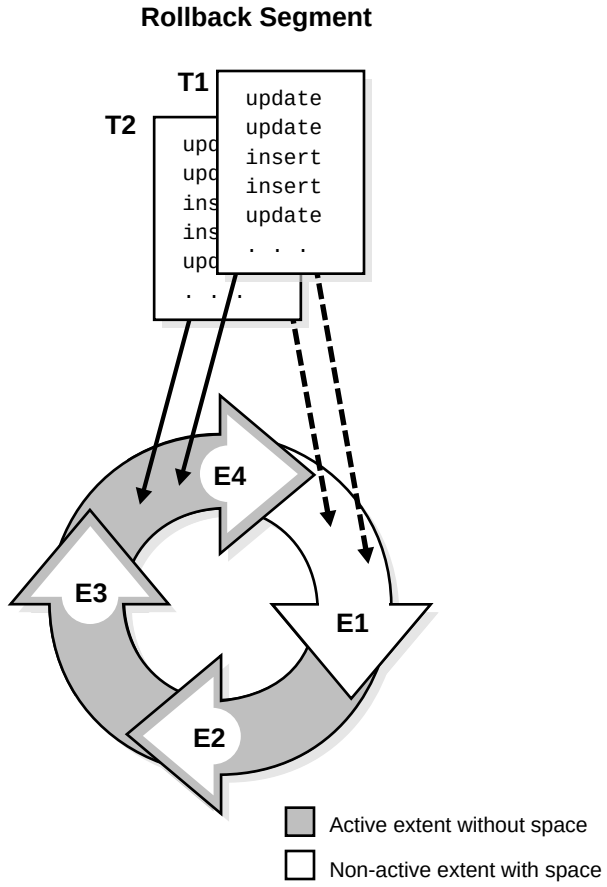
Figure 4–6 illustrates two transactions, T1 and T2, which begin writing in the third extent (E3) and continue writing to the fourth extent (E4) of a rollback segment.

Figure 4–6 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 4–7, 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 4–7 Cyclical Use of the Allocated Extents in a Rollback Segment

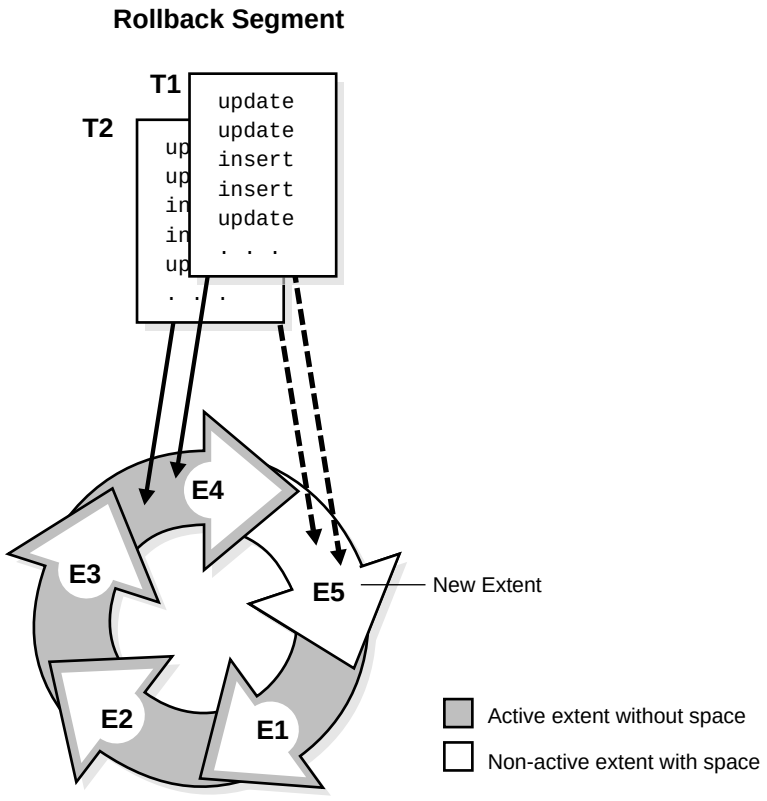


To continue writing rollback information for a transaction, Oracle always tries to reuse the next extent in the ring first. However, if the next extent contains data from active transaction, then Oracle must allocate a new extent. Oracle can allocate new extents for a rollback segment until the number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

Figure 4–8 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 4–8 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 non-SYSTEM rollback segments, Oracle uses the SYSTEM segment as necessary. In general, after database creation, you should 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 Oracle Parallel Server 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 example above, if TRANSACTIONS equal 155 and TRANSACTIONS_PER_ROLLBACK_SEGMENT equal 10, then the instance will try 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 above.)

   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 an Oracle Parallel Server 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 the Oracle Parallel Server optionally can have only public and no private segments, as long as the number of segments in the database is high

enough to ensure that each instance that opens the database can acquire at least two rollback segments, one of which is the SYSTEM rollback segment. However, when using the Oracle Parallel Server, you may want to use private rollback segments.

See Also:

- *Oracle8i Parallel Server Concepts*
- *Oracle8i Parallel Server Administration, Deployment, and Performance*

for more information about rollback segment use in an Oracle Parallel Server

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.

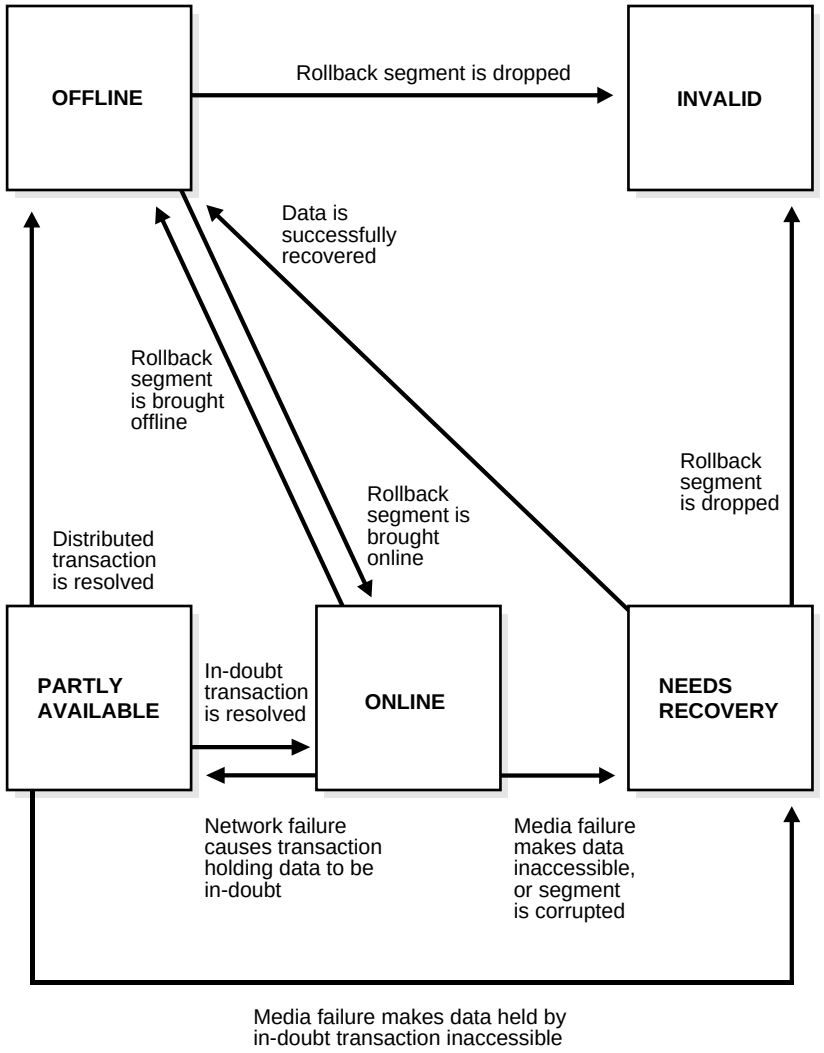
The rollback segment states are:

OFFLINE	Has not been acquired (brought online) by any instance.
ONLINE	Has been acquired (brought online) by an instance; may 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 4–9 shows how a rollback segment moves from one state to another.

Figure 4–9 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 allows the DBA to 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.

Also, 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 might need to acquire new extents for the active transactions, and therefore grow. To prevent the rollback segment from growing, a database administrator might prefer to 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:

- *Oracle8i Distributed Database Systems* for information about failures in distributed transactions
- "When Rollback Information Is Required" on page 4-22 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.

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:

5

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

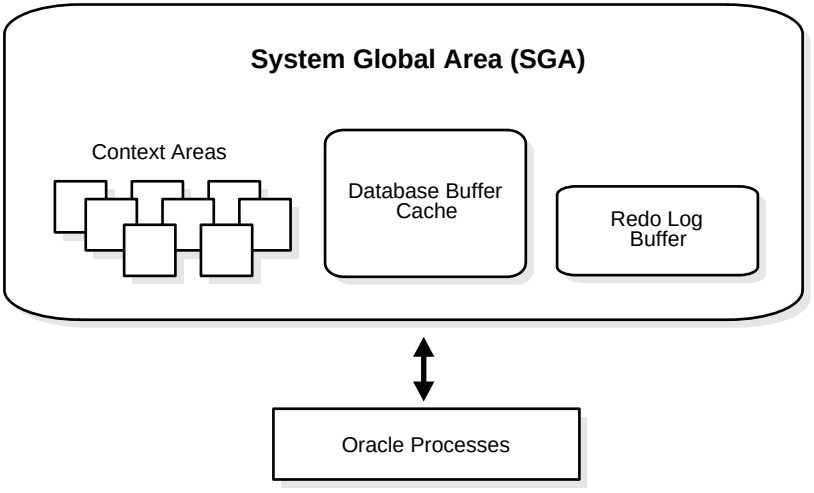
Every running Oracle database is associated with an Oracle instance. When a database is started on a database server (regardless of the type of computer), Oracle allocates a memory area called the System Global Area (SGA) and starts one or more Oracle processes. This combination of the SGA and the Oracle 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 5–1 shows an Oracle instance.

See Also:

- Chapter 7, "Memory Architecture"
- Chapter 8, "Process Architecture"

Figure 5–1 An Oracle Instance



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), the Oracle Parallel Server allows 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 via connections to Oracle with administrator privileges. Normal users do not have control over the current status of an Oracle database.

> **See Also:** *Oracle8i Parallel Server Concepts* for information about the Oracle Parallel Server.

Connecting 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.
- The database has a password for the INTERNAL login, and the user knows the password.

When you connect with SYSDBA or SYSOPER privileges, you are placed in the schema owned by SYS. 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 26, "Controlling Database Access" for more information about password files and authentication schemes for database administrators.

Parameter Files

To start an instance, Oracle must read a *parameter file*—a text file containing a list of configuration parameters (*initialization parameters*) for that instance and database. You 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 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
# rollback_segments = rs_one, rs_two
```

Changing Parameter Values

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.

Modified parameter values take effect only when the instance starts up and reads the parameter file. Some parameters can also be changed *dynamically* by using the ALTER SESSION or ALTER SYSTEM statement while the instance is running.

See Also:

- *Oracle8i Reference* for descriptions of all initialization parameters
- "Size of the SGA" on page 7-13 for information about parameters that affect the SGA

NLS Parameters

Oracle treats string literals defined for National Language Support (NLS) parameters in the file as if they are in the database character set.

See Also: *Oracle8i National Language Support Guide* for more information about National Language Support.

Instance and Database Startup

The three steps to starting a 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 Oracle Enterprise Manager.

See Also: *Oracle Enterprise Manager Administrator's Guide*

Starting an Instance

When Oracle starts an instance, first it reads a 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 7, "Memory Architecture" for information about the SGA
- Chapter 8, "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.

Forcing an Instance to 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 might 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.

Mounting a Database

The instance mounts a database to associate the database with that instance. After mounting 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.

Mounting a Database with the Oracle Parallel Server

Note: The features described in this section are available only if you have purchased Oracle8*i* Enterprise Edition with the Parallel Server Option.

See *Getting to Know Oracle8i* for more information.

If Oracle allows multiple instances to mount the same database concurrently, the database administrator can use the initialization parameter PARALLEL_SERVER to make the database available to multiple instances. The default value of the PARALLEL_SERVER parameter is FALSE. Versions of Oracle that do not support the Parallel Server Option only allow PARALLEL_SERVER to be FALSE.

If PARALLEL_SERVER is FALSE for the first instance that mounts a database, only that instance can mount the database. If PARALLEL_SERVER is set to TRUE on the first instance, other instances can mount the database if their PARALLEL_SERVER 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:

- *Oracle8i Parallel Server Administration, Deployment, and Performance*
- *Oracle8i Parallel Server Setup and Configuration Guide*

for more information about the use of multiple instances with a single database

Mounting a Standby Database

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:

- "Survivability" on page 29-27 for more information about standby databases and disaster recovery
- "Opening a Database in Read-Only Mode" on page 5-9 for information about opening a standby database in read-only mode

Mounting a Clone Database

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: *Oracle8i Backup and Recovery Guide* for detailed information about clone databases and tablespace point-in-time recovery

Opening 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 3-9 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.

> **See Also:** "Database Instance Failure" on page 29-4

Rollback Segment Acquisition

When you open the database, the instance attempts to acquire one or more rollback segments.

> **See Also:** "The Rollback Segment SYSTEM" and "Oracle Instances and Types of Rollback Segments" on page 4-27

Resolution of In-Doubt Distributed Transaction

Occasionally a database may close 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:**
>
> - Chapter 30, "Distributed Database Concepts"
> - *Oracle8i Distributed Database Systems*
>
> for information about recovery from distributed transaction failures

Opening 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.

In an Oracle Parallel Server, all instances must open the database either in read-write mode or in read-only mode.

> **See Also:** *Oracle8i 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. Dismount 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*

Closing 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 will 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.

Closing 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.

Dismounting a Database

Once the database is closed, Oracle dismounts the database to disassociate it from the instance. At this point, the instance remains in the memory of your computer.

After a database is dismounted, Oracle closes the control files of the database.

Shutting 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:** *Oracle8i Administrator's Guide* for more detailed information about instance and database startup and shutdown

6

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:

Introduction to Oracle 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 via a network. The following sections discuss possible variations in the Oracle client / server architecture.

Distributed Processing

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 6–1.

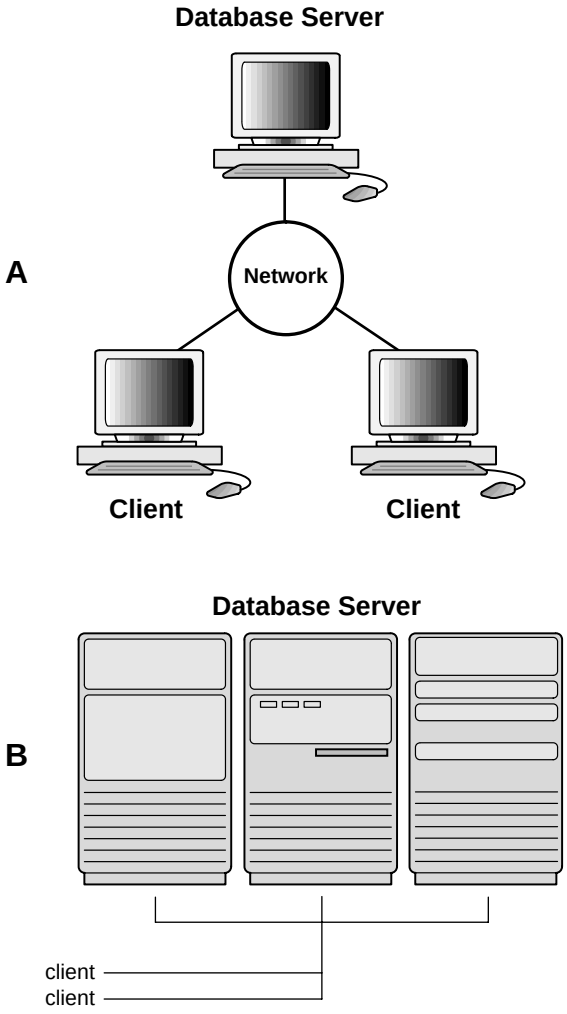
- In Part A of the figure, the client and server are located on different computers; these computers are connected via a network. The server and clients of an Oracle database system communicate via Net8, Oracle's network interface.
- In Part B of the figure, a single computer has more than one processor, 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:

- "Net8" on page 6-5 for more information about Net8
- Chapter 30, "Distributed Database Concepts", for more information about clients and servers in distributed databases

Figure 6–1 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, since 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.

Net8

Net8 is the Oracle network interface that allows Oracle tools running on network workstations and servers to access, modify, share, and store data on other servers. Net8 is considered part of the program interface in network communications.

Net8 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 standards, implemented in software, that govern the transmission of data across a network.
- An API is a set of subroutines that provide, in the case of networks, a means to establish remote process-to-process communication via a communication protocol.

Communication protocols define the way that data is transmitted and received on a network. In a networked environment, an Oracle server communicates with client workstations and other Oracle servers using Net8. Net8 supports communications on all major network protocols, ranging from those supported by PC LANs to those used by the largest mainframe computer systems.

Without the use of Net8, an application developer must manually code all communications in an application that operates in a networked distributed processing environment. If the network hardware, topology, or protocol changes, the application has to be modified accordingly.

However, by using Net8, the application developer does not have to be concerned with supporting network communications in a database application. If the underlying protocol changes, the database administrator makes some minor changes, while the application requires no modifications and will continue to function.

> **See Also:** Chapter 8, "Process Architecture" for more information about the program interface

How Net8 Works

Net8 drivers provide an interface between Oracle processes running on the database server and the user processes of Oracle tools running on other computers of the network.

The Net8 drivers take SQL statements from the interface of the Oracle tools and package them for transmission to Oracle via one of the supported industry-

standard higher level protocols or programmatic interfaces. The drivers also take replies from Oracle and package them for transmission to the tools via 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 Net8 software of the database server may include the driver software and start an additional Oracle background process.

See Also:

- Your Oracle operating system-specific documentation for details about the behavior of Net8
- *Net8 Administrator's Guide*

The Network Listener

When an instance starts, a *network 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 process or a dedicated server process and establishes an appropriate connection.

The listener process also establishes a communication pathway between databases. When multiple databases or instances run on one machine, as in an Oracle Parallel Server, *service names* allow 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.

Automatic instance registration reduces the administrative overhead for multiple databases or instances. The system identifiers (SIDs) of other instances on the network must be registered in a LISTENER.ORA file.

The initialization parameter SERVICE_NAMES identifies which 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:

- "Multi-Threaded Server Configuration" on page 8-16for more information about server processes
- "Dedicated Server Configuration" on page 8-23 for more information about server processes
- *Net8 Administrator's Guide* for more information about the network listener
- *Oracle8i Parallel Server Setup and Configuration Guide* and *Oracle8i Parallel Server Administration, Deployment, and Performance* for information about instance registration and client / service connections in an Oracle Parallel Server

Multi-Tier Architecture

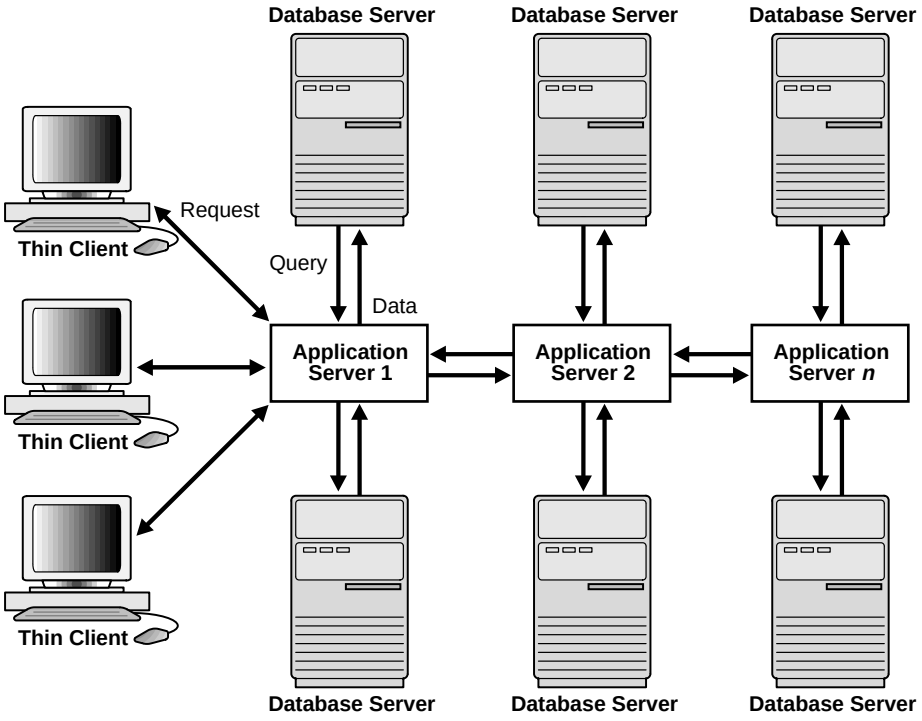
In a multi-tier architecture environment, an application server provides data for clients and serves as an interface between clients and database servers.

This architecture allows you to use 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 multi-tier architecture appears in Figure 6–2.

Figure 6–2 A Multi-Tier 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 multi-tier 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 might be a request for information to be displayed on the client whereas an application server operation might be a request for a connection to the database server.

> **See Also:** "Multi-Tier Authentication and Authorization" on page 26-9 for more information about security issues in a multi-tier environment

7

Memory Architecture

This chapter discusses the memory architecture of an Oracle instance. It includes:

- Introduction to Oracle Memory Structures
- System Global Area (SGA)
- Program Global Areas (PGA)
- Sort Areas
- Virtual Memory
- 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:

- Software Code Areas
- System Global Area (SGA):
  - The database buffer cache
  - The redo log buffer
  - The shared pool
- Program Global Areas (PGA):
  - The stack areas
  - The data areas

System Global Area (SGA)

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 referred to as 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 may 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:

- The database buffer cache
- The redo log buffer
- The shared pool
- The large pool (optional)
- The 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 multi-threaded server architecture, then the request and response queues and some contents of the program global areas, are in the SGA.

See Also:

- "Introduction to an Oracle Instance" on page 5-2 for more information about an Oracle instance
- "Program Global Areas (PGA)" on page 7-15
- "Dispatcher Request and Response Queues" on page 8-17

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 *least recently used (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 towards 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 (DBWn)" on page 8-8 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 may want to specify this behavior for small lookup tables or large static historical tables to avoid I/O on subsequent accesses of the table.

> **See Also:** *Oracle8i SQL Reference* for information on the CACHE clause

Size of the Database Buffer Cache

The initialization parameter DB_BLOCK_BUFFERS specifies the number of buffers in the database buffer cache. Each buffer in the cache is the size of one Oracle data block (which is specified by the initialization parameter DB_BLOCK_SIZE); therefore, each database buffer in the cache can hold a single data block read from a datafile.

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.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for more information on 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.

See Also:

- *Oracle8i Designing and Tuning for Performance* for more information on buffer pools
- *Oracle8i 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 8-9 for more information about how the redo log buffer is written to disk
- *Oracle8i Backup and Recovery Guide* for information about online redo log files and groups

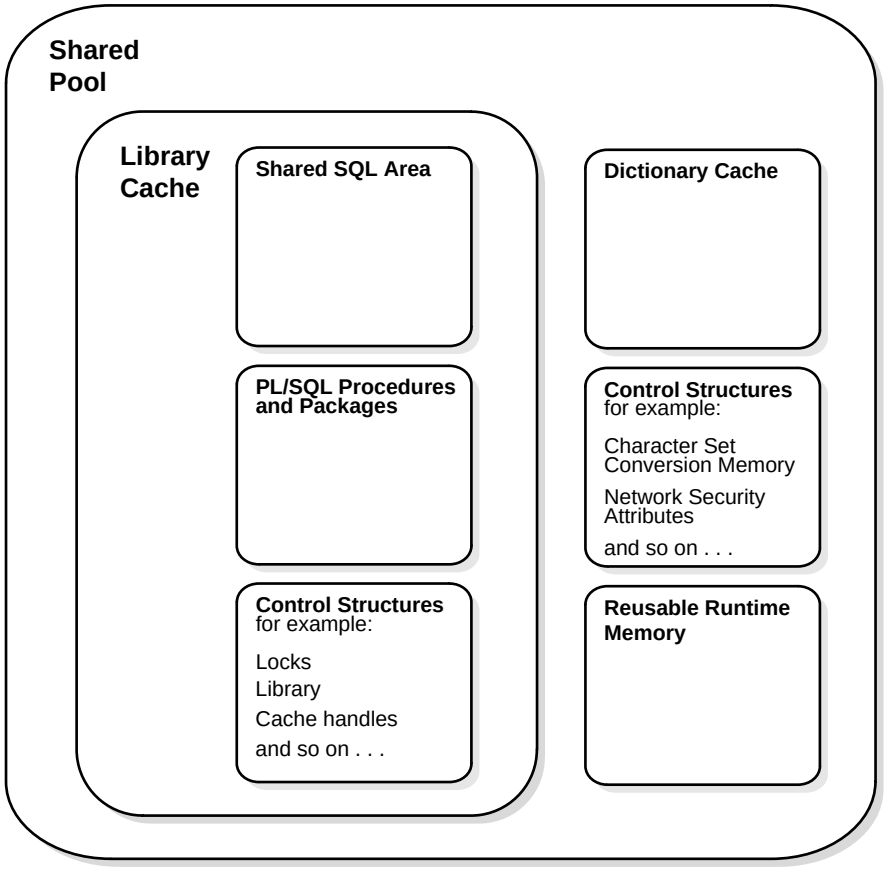
The initialization parameter LOG_BUFFER determines the size (in bytes) of the redo log buffer. In general, larger values reduce log file I/O, particularly if transactions are long or numerous. The default setting is four times the maximum data block size for the host operating system.

The Shared Pool

The shared pool portion of the SGA contains three major areas: library cache, dictionary cache, and control structures. Figure 7–1 shows the contents of the shared pool.

The total size of the shared pool is determined by the initialization parameter SHARED_POOL_SIZE. The default value of this parameter is 3,500,000 bytes. Increasing the value of this parameter increases the amount of memory reserved for the shared pool, and therefore increases the space reserved for shared SQL areas.

Figure 7–1 Contents of the Shared Pool



Library Cache

The library cache includes the shared SQL areas, private SQL areas, PL/SQL procedures and packages, and control structures such as locks and library cache handles.

Shared SQL areas must be available to multiple users, so the library cache is contained in the shared pool within the SGA. The size of the library cache (along with the size of the data dictionary cache) is limited by the size of the shared pool.

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 single SQL statement, or for similar SQL statements. Oracle saves memory by using one shared SQL area for multiple similar DML statements, particularly when many users execute the same application. A shared SQL area is always in the shared pool.

Oracle allocates memory from the shared pool when a SQL statement is parsed; the size of this memory depends on the complexity of the statement. If a SQL statement requires a new shared SQL area and the entire shared pool has already been allocated, Oracle can deallocate items from the pool using a modified 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 when it is next executed.

> **See Also:** *Oracle8i Designing and Tuning for Performance*

Private SQL Areas A private SQL area contains data such as bind information and runtime buffers. Each session that issues a SQL statement has a private SQL area. Each user that submits an identical SQL statement has his or her own private SQL area that uses a single shared SQL area; many private SQL areas can be associated with the same shared SQL area.

A private SQL area has a persistent area and a runtime area:

- The *persistent area* contains bind information that persists across executions, code for datatype conversion (in case the defined datatype is not the same as the datatype of the selected column), and other state information (like recursive or remote cursor numbers or the state of a parallel query). The size of the

persistent area depends on the number of binds and columns specified in the statement. For example, the persistent area is larger if many columns are specified in a query.

- The *runtime area* contains information used while the SQL statement is being executed. The size of the runtime area depends on the type and complexity of the SQL statement being executed and on the sizes of the rows that are processed by the statement. In general, the runtime area is somewhat smaller for INSERT, UPDATE, and DELETE statements than it is for SELECT statements, particularly when the SELECT statement requires a sort.

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 via a dedicated server, private SQL areas are located in the user's PGA. However, if a session is connected via the multi-threaded server, the persistent areas and, for SELECT statements, the runtime areas, are kept in the SGA.

See Also:

- "Connections and Sessions" on page 8-4 for more information about sessions
- "Sort Areas" on page 7-17 for information about SELECT runtimes during a sort

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

should 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.

For queries that process large amounts of data requiring sorts, application developers should cancel the query if a partial result of a fetch is satisfactory. For example, in an email application, a user can select from a list of templates for creating a mail message. When the email application displays the first ten template names, if the user chooses one of these templates, then the application should cancel the processing of the rest of the query, rather than continue trying to display more template names.

See Also: "Cursors" on page 15-7

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 the parsing of SQL statements. 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 2, "The Data Dictionary"
- "Library Cache" on page 7-8

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 exact location of the private SQL area depends on the connection established for a 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.

See Also:

- "Shared SQL Areas and Private SQL Areas" on page 7-8for more information about the location of the private SQL area
- Chapter 20, "Oracle Dependency Management" for more information about the invalidation of SQL statements and dependency issues

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 multi-threaded server and the Oracle XA interface
- I/O server processes
- Oracle backup and restore operations

By allocating session memory from the large pool for the multi-threaded server or for Oracle XA, Oracle can use the shared pool primarily for caching shared SQL and avoid the performance overhead caused by shrinking the shared SQL cache.

The memory for Oracle backup and restore operations and for I/O server processes is allocated in buffers of a few hundred kilobytes. The large pool is better able to satisfy such 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:

- "Multi-Threaded Server Configuration" on page 8-16 for information about allocating session memory from the large pool for the multi-threaded server
- *Oracle8i Application Developer's Guide - Fundamentals* for information about Oracle XA
- *Oracle8i Designing and Tuning for Performance* for more information about the large pool, reserve space in the shared pool, and I/O server processes

Size of the SGA

The size of the SGA is determined at instance start up. For optimal performance in most systems, the entire SGA should fit in real memory. If it does not fit in real memory, 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_BLOCK_SIZE	The size, in bytes, of a single data block and database buffer.
DB_BLOCK_BUFFERS	The number of database buffers, each the size of DB_BLOCK_SIZE, allocated for the SGA. The total amount of space allocated for the database buffer cache in the SGA is DB_BLOCK_SIZE times DB_BLOCK_BUFFERS.
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.

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:

- "Virtual Memory" on page 7-18
- *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
- *Oracle8i Designing and Tuning for Performance* for discussions of the above initialization parameters and how they affect the SGA
- Your Oracle installation or user's guide for information specific to your operating system

Controlling the SGA's Use of Memory

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 used 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.

See Also:

- *Oracle8i 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)

A *program global area* (PGA) is a memory region containing data and control information for a single process (server or background). Consequently, a PGA is sometimes called a *process global area*.

A PGA is nonshared memory area to which a process can write. One PGA is allocated for each server process; the PGA is exclusive to that server process and is read and written only by Oracle code acting on behalf of that process.

A PGA is allocated by Oracle when a user connects to an Oracle database and a session is created, though this varies by operating system and configuration.

See Also: "Connections and Sessions" on page 8-4 for information about sessions

Contents of a PGA

The contents of a PGA vary, depending on whether the associated instance is running the multi-threaded server.

See Also: "Multi-Threaded Server Configuration" on page 8-16 for more information on the multi-threaded server

Stack Space

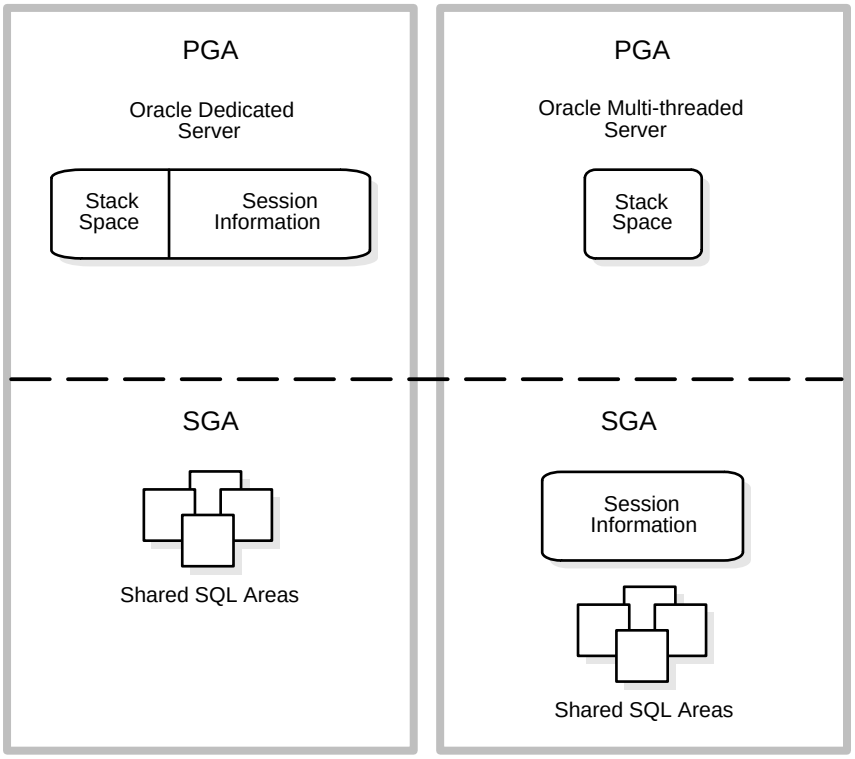
A PGA always contains a *stack space*, which is memory allocated to hold a session's variables, arrays, and other information.

Session Information

If the instance is running without the multi-threaded server, the PGA also contains information about the user's session, such as private SQL areas. If the instance is running in multi-threaded server configuration, this session information is not in the PGA, but is instead allocated in the SGA.

Figure 7–2 shows where the session information is stored in different configurations.

Figure 7–2 Location of Session Information With and Without Multi-threaded Server



Size of a PGA

A PGA's initial size is fixed and operating-system specific. When the client and server are on different machines, the PGA is allocated on the database server at connect time; if sufficient memory is not available to connect, an Oracle error occurs with an error number in the range for that operating system. Once connected, a user can never run out of PGA space; there is either enough or not enough memory to connect in the first place.

The initialization parameters OPEN_LINKS and DB_FILES affect the sizes of PGAs. The size of the stack space in each PGA created on behalf of Oracle background processes (such as DBW0 and LGWR) is affected by some additional parameters.

> **See Also:** Your Oracle operating system-specific documentation for more information about the PGA

Sort Areas

Sorting requires space in memory. Portions of memory in which Oracle sorts data are called *sort areas*. Sort areas use memory from the PGA of the Oracle server process that performs the sort on behalf of the user process. However, a part of the sort area (up to SORT_AREA_RETAINED_SIZE) exists in the runtime area of the process's private SQL area. For SELECT statements this memory in the private SQL area comes from different places depending on the connection configuration:

- From the PGA for connections through a dedicated server
- From the SGA for connections through the multi-threaded server.

A sort area can grow to accommodate the amount of data to be sorted but is limited by the value of the initialization parameter SORT_AREA_SIZE. The default value, expressed in bytes, is operating system specific.

During a sort, Oracle may perform some tasks that do not involve referencing data in the sort area. In such cases, Oracle may decrease the size of the sort area by writing some of the data to a temporary segment on disk and then deallocating the portion of the sort area that contained that data. Such deallocation may occur, for example, if Oracle returns control to the application.

The initialization parameter SORT_AREA_RETAINED_SIZE determines the size to which the sort area is reduced. The default value of this parameter is the value of the SORT_AREA_SIZE parameter.

Memory released during a sort is freed for use by the same Oracle process, but it is not released to the operating system.

If the amount of data to be sorted does not fit into a sort area, then the data is divided into smaller pieces that do fit. Each piece is then sorted individually. The individual sorted pieces are called *runs*. After sorting all the runs, Oracle merges them to produce the final result.

See Also:

- "Private SQL Areas" on page 7-8 for more information about where the private SQL area memory comes from
- "Dedicated Server Configuration" on page 8-23 for more information about the private SQL area in a dedicated server configuration
- "Multi-Threaded Server Configuration" on page 8-16 for more information about the private SQL area in a multi-threaded server configuration

Virtual Memory

On many operating systems, Oracle takes advantage of *virtual memory*—an operating system feature that offers more apparent memory than is provided by real memory alone and more flexibility in using main memory.

Virtual memory simulates memory using a combination of real (main) memory and secondary storage (usually disk space). The operating system accesses virtual memory by making secondary storage look like main memory to application programs.

Suggestion: Usually, it is best to keep the entire SGA in real memory. On many platforms, you can lock the SGA into real memory with the LOCK_SGA parameter.

Software Code Areas

Software code areas are portions of memory used to store code that is being executed or may 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 may 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 MS-DOS).
>
> See your Oracle operating system-specific documentation for more information.

8

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
- Oracle Processes
- Multi-Threaded Server Configuration
- 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:

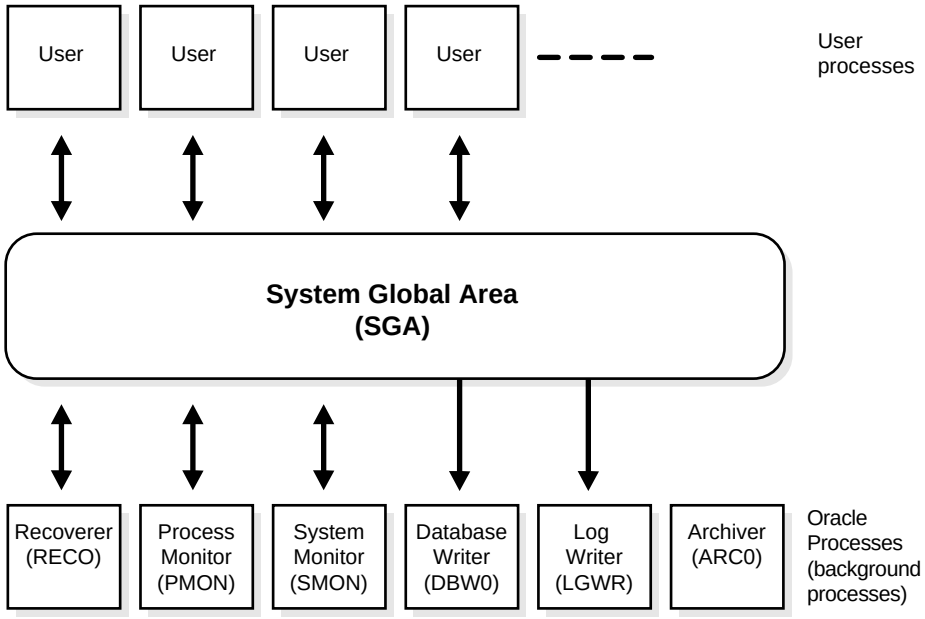
- User processes execute the application or Oracle tool code.
- Oracle processes execute the Oracle server code. They include server processes and background processes.

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).
multi-threaded 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.
pre-spawned dedicated server	For each user, the database application is run by a different user process than the one that executes the Oracle server code. The user process is started as part of the configuration before the user requests it.

Figure 8–1 illustrates a dedicated server configuration. Each connected user has a separate user process, and several background processes execute Oracle.

Figure 8–1 A Multiple-Process Oracle Instance



This figure might 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" on page 8-4
- "Oracle Processes" on page 8-5
- "Dedicated Server Configuration" on page 8-23
- "Multi-Threaded Server Configuration" on page 8-16
- Your Oracle operating system-specific documentation for more details on choices of configuration

User Processes

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 via a network).

A *session* is a specific connection of a user to an Oracle instance via 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 multi-threaded server, Oracle creates a server process on behalf of each user session; however, with the multi-threaded server, many user sessions can share a single server process.

See Also: "Multi-Threaded Server Configuration" on page 8-16

Oracle Processes

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 via a separate server process.

Server processes (or the server portion of combined user/server processes) created on behalf of each user's application may perform one or more of the following:

- Parse and execute SQL statements issued via 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 may 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 (LCK0)
- Job Queue (SNP*n*)
- Queue Monitor (QMN*n*)
- Dispatcher (D*nnn*)
- Server (S*nnn*)

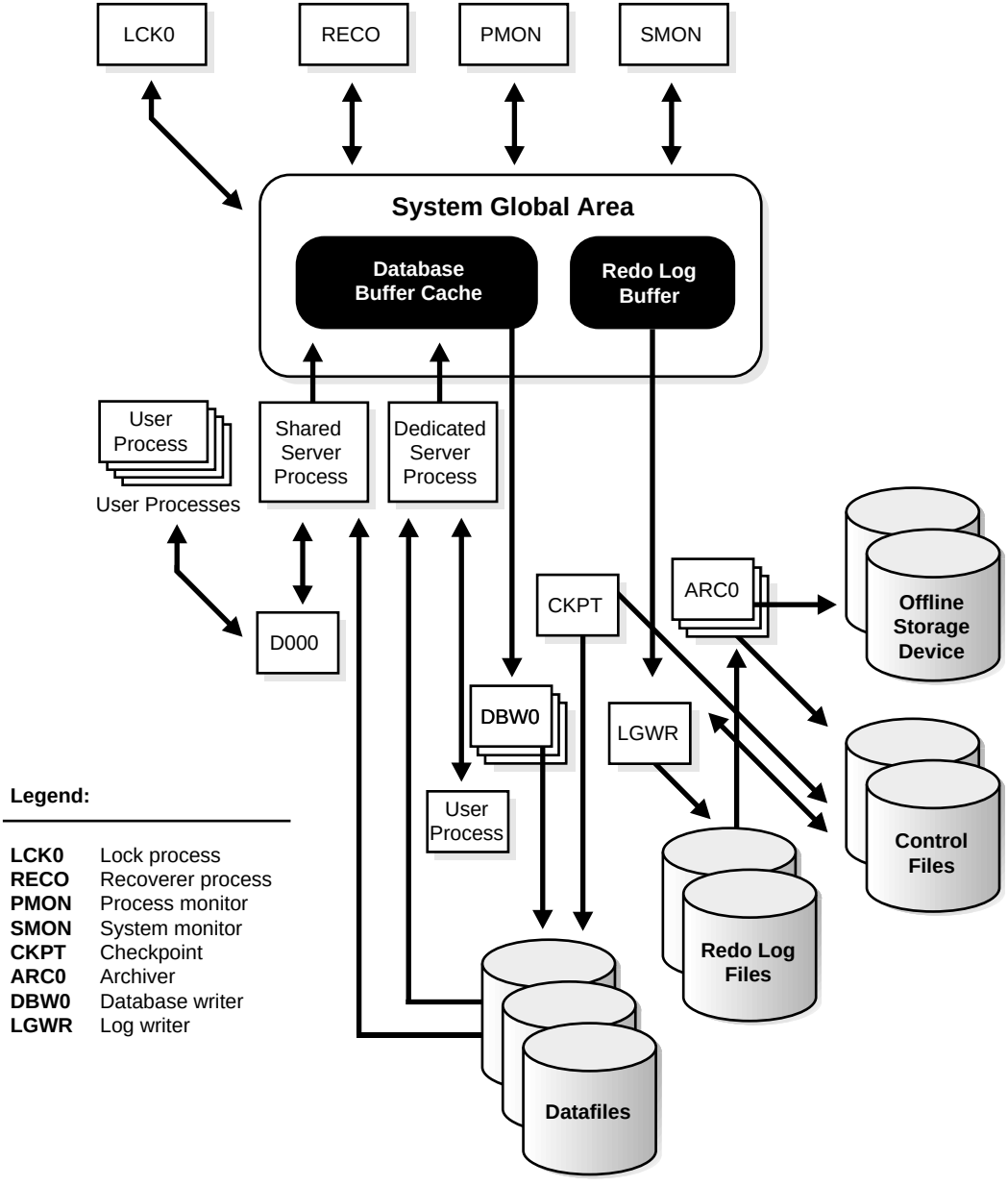
On many operating systems, background processes are created automatically when an instance is started.

Figure 8–2 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:

- *Oracle8i Parallel Server Concepts* for more information; the Oracle Parallel Server is not illustrated in Figure 8–2
- Your Oracle operating system-specific documentation for details on how these processes are created

Figure 8–2 The Background Processes of a Multiple-Process Oracle Instance



Database Writer (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 will be 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 may be useful soon.

The initialization parameter DB_WRITER_PROCESSES specifies the number of DBW*n* processes. If your system uses multiple DBW*n* processes, you should 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 would need to begin. 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 7-3
- *Oracle8i Designing and Tuning for Performance* for advice on setting DB_WRITER_PROCESSES and DB_BLOCK_LRU_LATCHES
- "Fast-Start Checkpointing" on page 29-14
- *Oracle8i Designing and Tuning for Performance* for information about how to monitor and tune the performance of a single DBW0 process or multiple DBW*n* processes

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, even though 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 Oracle Parallel Server configurations and distributed databases.

In times of high activity, LGWR may 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 would 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 may contain multiple commit records.

See Also:

- "The Redo Log Buffer" on page 7-6
- "Trace Files and the ALERT File" on page 8-15
- *Oracle8i Parallel Server Administration, Deployment, and Performance* for more information about SCNs and how they are used
- *Oracle8i Administrator's Guide* for more information about SCNs and how they are used
- *Oracle8i Designing and Tuning for Performance* for 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: *Oracle8i Parallel Server Administration, Deployment, and Performance* for information about CKPT in an Oracle Parallel Server environment.

System Monitor (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.

In an Oracle Parallel Server environment, the SMON process of one instance can perform instance recovery for a failed CPU or instance.

See Also: *Oracle8i Parallel Server Administration, Deployment, and Performance* for more information about SMON

Process Monitor (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:** *Oracle8i Distributed Database Systems* 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.

Archiver Processes (ARC*n*)

The *archiver process (ARCn)* copies online redo log files to a designated storage device once they become full or when the ALTER SYSTEM SWITCH LOGFILE statement forces a log switch. 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 ten 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:

- "Database Archiving Modes" on page 29-18
- "Trace Files and the ALERT File" on page 8-15
- "The Redo Log" on page 29-7 for information on archiving the online redo log
- *Oracle8i Backup and Recovery Guide*
- Your Oracle operating system-specific documentation for details about using the ARC*n* processes

Lock Process (LCK0)

In an Oracle Parallel Server, a *lock process* (LCK0) provides inter-instance locking.

See Also: *Oracle8i Parallel Server Administration, Deployment, and Performance* for more information about this background process

Job Queue Processes (SNP*n*)

With the distributed database configuration, up to thirty-six *job queue processes* (SNP0, ..., SNP9, SNPA, ..., SNPZ) can automatically refresh table snapshots. These processes wake up periodically and refresh any snapshots that are scheduled to be automatically refreshed. If more than one job queue process is used, the processes share the task of refreshing snapshots.

Unlike other Oracle background processes, failure of an SNP*n* process does not cause the instance to fail. If an SNP*n* process fails, Oracle restarts it.

These processes also execute job requests created by the DBMS_JOB package and propagate queued messages to queues on other databases.

See Also:

- "Oracle Advanced Queuing" on page 18-3
- *Oracle8i Administrator's Guide* for more information about this background process and job queues

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 ten queue monitor processes. These processes, like the SNP*n* processes, are different from other Oracle background processes in that process failure does not cause the instance to fail.

See Also: "Oracle Advanced Queuing" on page 18-3

Dispatcher Processes (D*nnn*)

The *dispatcher processes* support multi-threaded configuration by allowing user processes to share a limited number of server processes. With the multi-threaded server, fewer shared server processes are required for the same number of users; therefore, the multi-threaded 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 should start an optimal number of dispatcher processes depending on the operating system limitation on the number of connections per 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 Net8 or SQL*Net version 2, even if both processes are running on the same machine.

In a multi-threaded 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:

- "Multi-Threaded Server Configuration" on page 8-16
- *Net8 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 multi-threaded server configuration.

See Also: "Shared Server Processes" on page 8-20

Trace Files and the ALERT File

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 (SNP*n*).

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 file*. The ALERT file of a database is a chronological log of messages and errors, including

- 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 snapshot

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 timestamp.

See Also:

- *Oracle8i Designing and Tuning for Performance* for information about enabling the SQL trace facility
- *Oracle8i Error Messages* for information about error messages

Multi-Threaded Server Configuration

The multi-threaded server configuration allows many user processes to share very few server processes. The user processes connect to a dispatcher background process, which routes client requests to the next available shared server process.

The advantage of the multi-threaded server configuration is that system overhead is reduced, increasing the number of users that can be supported. A small number of shared server processes can perform the same amount of processing as many dedicated server processes, and the amount of memory required for each user is relatively small.

A number of different processes are needed in a multi-threaded server system:

- A network listener process that connects the user processes to dispatchers or dedicated servers (the listener process is part of Net8, not Oracle).
- One or more dispatcher processes
- One or more shared server processes

The multi-threaded server requires Net8 or SQL*Net version 2.

Note: To use shared servers, a user process must connect through Net8 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 (such as those that connect using pre-version 2 SQL*Net) 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 Multi-Threaded Server" on page 8-21
- *Net8 Administrator's Guide* for more information about the network listener

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.

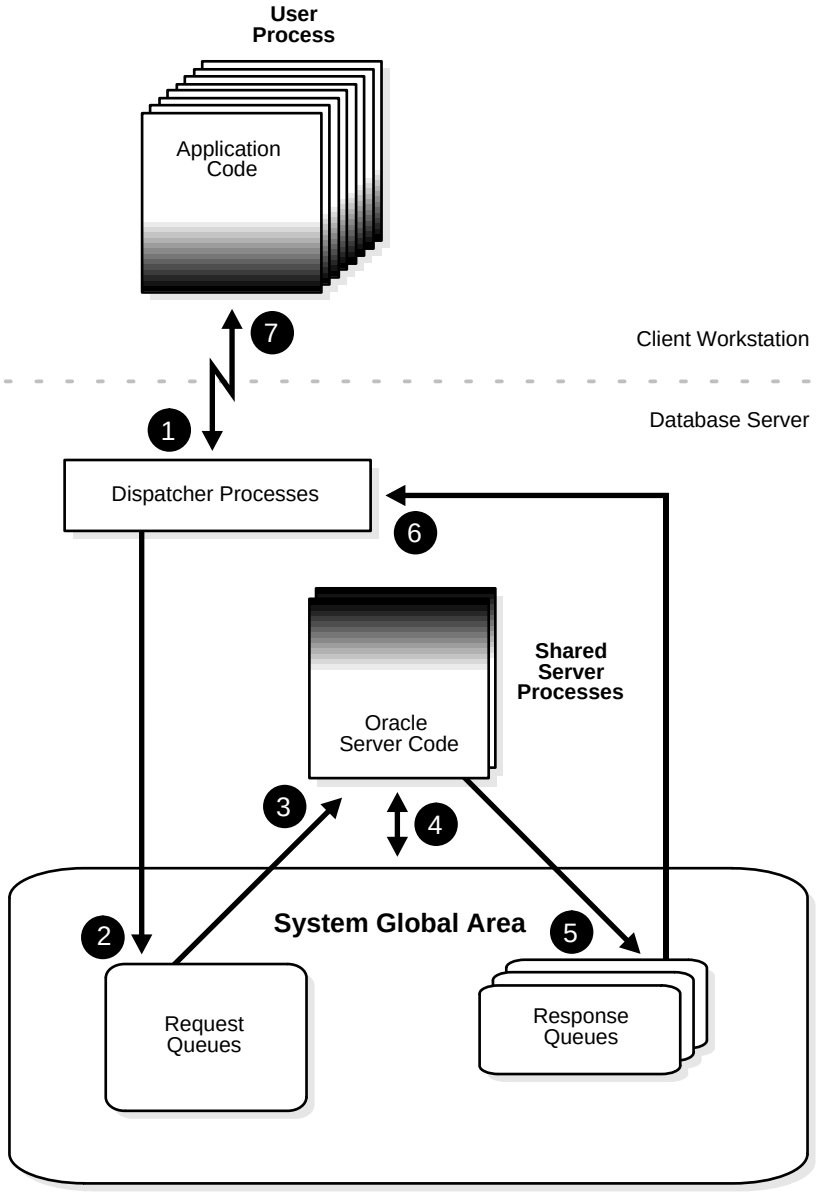
The request queue is in the SGA and is common to all dispatcher processes of an instance. The shared server processes check the common request queue for new requests, picking up new requests on a first-in-first-out basis. One shared server process picks up one request in the queue and makes all necessary 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 8–3 illustrates how user processes communicate with the dispatcher across the program interface and how the dispatcher communicates users' requests to shared server processes.

Figure 8–3 The Multi-Threaded Server Configuration and Shared Server Processes



Shared Server Processes

Shared server processes and dedicated server processes provide the same functionality, except that shared server processes are not associated with a specific user process. Instead, a shared server process serves any client request in the multi-threaded 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 MTS_SERVERS and MTS_MAX_SERVERS.

See Also:

- "Program Global Areas (PGA)" on page 7-15 for more information about the content of a PGA in different types of instance configurations
- Chapter 26, "Controlling Database Access"for more information about resource limits and profiles

Artificial Deadlocks

With a limited number of shared server processes, the possibility of an *artificial deadlock* can arise. An artificial deadlock can occur in the following situation:

1. One user acquires an exclusive lock on a resource by issuing a SELECT statement with the FOR UPDATE clause or a LOCK TABLE statement.
2. The shared server process that processes the locking request is released once the statement completes.
3. Other users attempt to access the locked resource. Each shared server process is bound to the user process it is serving until the necessary locked resource becomes available. Eventually, all shared servers may be bound to user processes waiting for locked resources.

4. The original user attempts to submit a new request (such as a COMMIT or ROLLBACK statement) to release the previously acquired lock, but cannot because all shared server processes are currently being used.

When Oracle detects an artificial deadlock, new shared server processes are automatically created as needed until the original user submits a request that releases the locked resources causing the artificial deadlocks. If the maximum number of shared server processes (as specified by the MTS_MAX_SERVERS parameter) have been started, the database administrator must manually resolve the deadlock by disconnecting a user. This releases a shared server process, resolving the artificial deadlock.

If artificial deadlocks occur too frequently on your system, you should increase the value of MTS_MAX_SERVERS.

Restricted Operations of the Multi-Threaded 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 multi-threaded servers, you must state in your connect string that you want to use a dedicated server process (SRVR=DEDICATED) instead of a dispatcher process.

See Also:

- Your Oracle operating system-specific documentation
- *Net8 Administrator's Guide*

for the proper connect string syntax

An Example of Oracle Using the Multi-Threaded Server

The following steps illustrate how Oracle works in the multi-threaded server configuration. These steps show only the most basic level of operations that Oracle performs.

1. A database server is currently running Oracle using the multi-threaded server configuration.

2. A user process on a client workstation runs a database application such as SQL*Forms. The client application attempts to establish a connection to the database server using the proper Net8 driver.
3. The database server machine is currently running the proper Net8 driver. The network listener process on the database server detects the connection request of the user process and determines how the user process should be connected. If the user is using Net8 or SQL*Net version 2, the listener informs the user process to reconnect using the address of an available dispatcher process.

   Note: If the user process connects with SQL*Net version 1 or 1.1, the SQL*Net listener creates a dedicated server process on behalf of the user process and the remainder of the example operates as described in the preceding example. (User processes must connect with Net8 or SQL*Net version 2 to use a shared server process.)

4. The user issues a single SQL statement, for example, updating a row in a table.
5. The dispatcher process places the user process's request on the request queue, which is in the SGA and shared by all dispatcher processes.
6. An available shared server process checks the common dispatcher request queue and picks up the next SQL statement on the queue. At this point, two paths can be followed to continue processing the SQL statement:
   - If the shared pool contains a shared SQL area for a similar SQL statement, the server process uses the existing shared SQL area to execute the client's SQL statement.
   - If the shared pool does not contain a shared SQL area for a similar SQL statement, a new shared SQL area is allocated for the statement in the shared pool.

   In either case, a private SQL area is created (partly in the session's PGA and partly in the SGA) and the shared server process checks the user's access privileges to the requested data.
7. The shared server process retrieves data blocks from the actual datafile, if necessary, or uses data blocks already stored in the buffer cache in the SGA of the instance.
8. The shared server process executes the SQL statement stored in the shared SQL area. Data is first changed in the SGA. It is permanently written to disk when the DBW0 process determines it is most efficient to do so. The LGWR process

records the transaction in the online redo log file only on a subsequent commit request from the user.

9. Once the shared server process finishes processing the SQL statement, the process places the result on the response queue of the dispatcher process that sent the request.

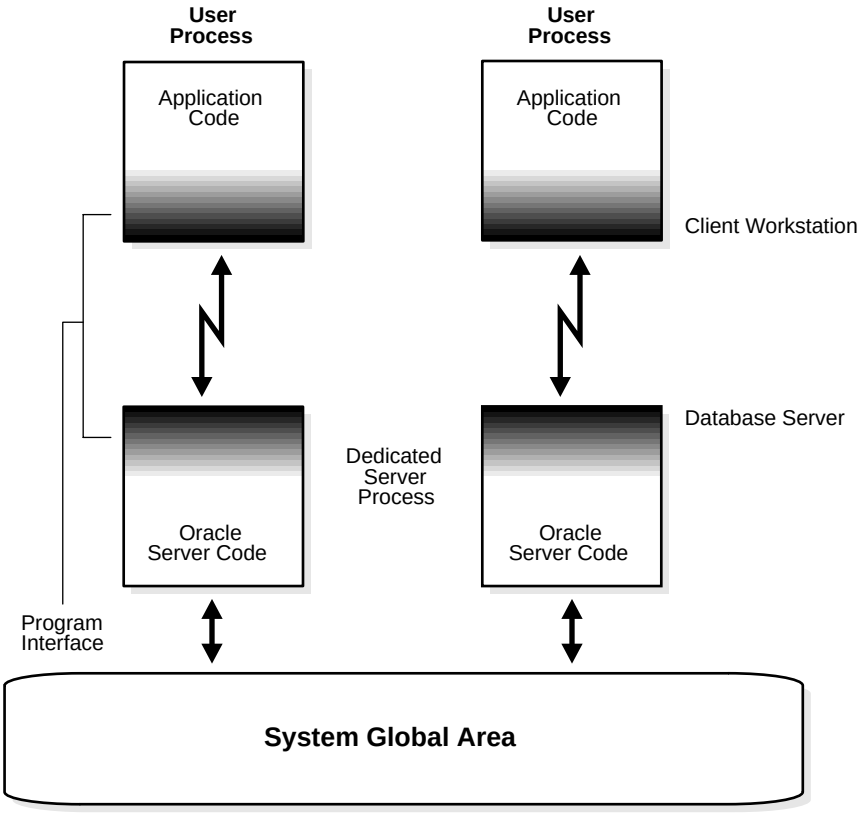
10. The dispatcher process checks its response queue and sends completed requests back to the user process that made the request.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for information about shared SQL areas

Dedicated Server Configuration

Figure 8–4 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 8–4 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 may be paged out on some operating systems).

Figure 8–4 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 Net8) 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 multi-threaded server architecture may be preferable.

See Also:

- Your Oracle operating system-specific documentation
- *Net8 Administrator's Guide*

for more information about communication links

An Example of Oracle Using Dedicated Server Processes

The following steps illustrate how Oracle works in the dedicated server configuration. These steps show only the most basic level of operations that Oracle performs.

1. A database server machine is currently running Oracle using multiple background processes.
2. A user process on a client workstation runs a database application such as SQL*Plus. The client application attempts to establish a connection to the server using a Net8 driver.

3. The database server is currently running the proper Net8 driver. The network listener process on the database server detects the connection request from the client database application and creates a dedicated server process on the database server on behalf of the user process.
4. The user executes a single SQL statement. For example, the user inserts a row into a table.
5. The dedicated server process receives the statement. At this point, two paths can be followed to continue processing the SQL statement:
   - If the shared pool contains a shared SQL area for a similar SQL statement, the server process uses the existing shared SQL area to execute the client's SQL statement.
   - If the shared pool does not contain a shared SQL area for a similar SQL statement, a new shared SQL area is allocated for the statement in the shared pool.

   In either case, a private SQL area is created in the session's PGA and the dedicated server process checks the user's access privileges to the requested data.
6. The server process retrieves data blocks from the actual datafile, if necessary, or uses data blocks already stored in the buffer cache in the SGA of the instance.
7. The server process executes the SQL statement stored in the shared SQL area. Data is first changed in the SGA. It is permanently written to disk when the DBW0 process determines it is most efficient to do so. The LGWR process records the transaction in the online redo log file only on a subsequent commit request from the user.
8. If the request is successful, the server sends a message across the network to the user. If it is not successful, an appropriate error message is transmitted.
9. Throughout this entire procedure, the other background processes are running and watching for any conditions that require intervention. In addition, Oracle is managing other transactions and preventing contention between different transactions that request the same data

See Also: *Oracle8i Designing and Tuning for Performance* for information about shared SQL areas

Pre-Spawned Dedicated Processes

Pre-spawned processes are user processes that are started before an application requests them. Pre-spawned processes can improve connect time with a dedicated server. This is particularly true of heavily loaded systems that do not use multi-threaded servers. If pre-spawned processes are enabled, then the listener can hand off the connection to an existing process with no wait time whenever a connection request arrives. Once the user is connected, the listener creates the next shadow process for the next connect.

Pre-spawned user processes are also useful in controlling resources or access to the system. You can cap the number of shadow processes that can be pre-spawned. After this limit is reached, all new connections come in as dedicated.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for information about configuring pre-spawned dedicated processes

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 *Net8 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.

Net8 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.

The Program Interface Drivers

Drivers are pieces of software that transport data, usually across a network. They perform operations like connect, disconnect, signal errors, and test for errors. Drivers are specific to a communications protocol. There is always a default driver.

You may 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 Net8 drivers.

See Also:

- Your system installation and configuration guide for details about choosing, installing, and adding drivers
- Your system Net8 documentation for information about selecting a driver at runtime while accessing Oracle
- *Net8 Administrator's Guide*

Operating System Communications Software

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 may 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

9

Database Resource Management

This chapter describes how you can use the Database Resource Manager to allocate resources to different groups of users. This chapter includes the following topics:

- Introduction to the Database Resource Manager
- Resource Consumer Groups and Resource Plans
- Resource Allocation Methods
- Resource Plan Directives
- Examples
- Using the Database Resource Manager

Note: The features of the Database Resource Manager described in this chapter are available only if you have purchased the Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for information about Oracle8*i* Enterprise Edition.

Introduction to the Database Resource Manager

The Database Resource Manager allows the database administrator to have more control over resource management than would normally be possible through operating system resource management alone. Improved resource management enables better application performance and availability. By using the Database Resource Manager, the database administrator can:

- Guarantee groups of users a minimum amount of processing resources, regardless of the load or number of users in other groups on the system.
- Distribute available processing resources by allocating percentages of CPU time to different users and applications. For example, in a data warehouse, a higher priority may be given to ROLAP applications than to batch jobs.
- Limit the degree of parallelism that a set of users can use.
- Configure an instance to use a particular plan for allocating resources. A database administrator can dynamically change the plan, for example, from a daytime setup to a nighttime setup, without having to shut down and restart the instance.

To use the Database Resource Manager, a database administrator defines the following items:

resource consumer groups	Groups of user sessions that have similar processing and resource use requirements.
resource plans	Means of allocating resources among the consumer groups.
resource allocation methods	Policies for allocating a particular resource. Resource allocation methods are used by both plans and consumer groups.
resource plan directives	Means of: ■ assigning consumer groups or subplans to resource plans ■ allocating resources among consumer groups in the plan by specifying parameters for each resource allocation method.

These items are described in detail in the following sections.

Resource Consumer Groups and Resource Plans

Resource consumer groups and resource plans provide methods of specifying how to allocate processing resources among different users. Currently, CPU is the only resource that is controlled at the level of resource consumer groups. Resource plans currently support control of two resources: CPU and degree of parallelism.

This section describes resource consumer groups and plans.

What Are Resource Consumer Groups?

To control resource consumption, you can assign user sessions to *resource consumer groups*. A resource consumer group defines a set of users who have similar requirements for resource use. A resource consumer group also specifies a resource allocation method for allocating CPU among sessions. Currently, the only resource allocation method for allocating CPU among sessions in a resource consumer group is the *round-robin* method.

Users can have the privilege of switching to different consumer groups. You can use PL/SQL procedures to switch resource consumer groups for a particular session, assuming you have the appropriate privileges.

There are two special consumer groups that cannot be modified or deleted:

- OTHER_GROUPS applies to all sessions that belong to a consumer group that is not part of the active plan schema. This group name always exists in the data dictionary. OTHER_GROUPS must have a resource plan directive specified in the schema of any active plan.
- DEFAULT_CONSUMER_GROUP applies to all sessions that have not been explicitly assigned to an initial consumer group.This group name always exists in the data dictionary.

There are two Oracle-provided consumer groups which you can modify, use as is, not use, or delete as appropriate for your environment:

- SYS_GROUP is the initial consumer group for the users SYS and SYSTEM.
- LOW_GROUP is provided for use in the SYSTEM_PLAN (see Table 9–2, "SYSTEM_PLAN Default Resource Plan"). Switch privilege is granted to PUBLIC for this group.

What Are Resource Plans?

Resource allocations are specified in a *resource plan*. Resource plans contain resource plan *directives*, which specify the resources that are to be allocated to each resource consumer group.

Conceptually, a session belongs to a resource consumer group, and this resource consumer group is used by a resource plan to determine allocation of processing resources.

You can use resource plans to:

- Group resource consumer groups and / or other resource plans together
- Allocate resources among those resource consumer groups or plans
- Specify a resource allocation. (Currently, the only resource allocation method for allocating CPU among resource consumer groups is the *emphasis* method. The only resource allocation method for limiting the degree of parallelism is the *absolute* method. Both of these methods are explained below.)

You can have multiple resource plans defined in the database, each allocating resources to resource consumer groups in different ways, making resource assignment flexible. However, only one plan can be active at a time. For example, you might define a daytime plan, a nighttime plan, and a weekend plan. Different instances of an Oracle Parallel Server can use different resource plans.

You can specify resource plans in a hierarchical fashion using *subplans*. Activating a plan also activates all of its subplans.

You can dynamically switch the top-level active plan while an instance is running. This enables you to define resource plans for different situations, and to change the plan depending upon the situation.

You can view resource plans and their associated attributes in the data dictionary view DBA_RSRC_PLANS. Each entry contains the following information:

- Plan name
- Number of plan directives
- Its CPU resource allocation method
- Its parallel degree limit method
- Comment
- Status (pending or active)
- Statement about whether the plan is mandatory

Following are sample resource plan entries:

```
PLAN   NUM_PLAN_D  CPU_METHOD  PARALLEL_DEGREE_LIMIT_MTH       COMMENTS                    STATUS  MANDATORY
------ ----------  ----------- ------------------------------  --------------------------  ------- ---------
MAILDB   3         EMPHASIS    PARALLEL_DEGREE_LIMIT_ABSOLUTE  Plan/method for mail users  ACTIVE  0
APPDB    3         EMPHASIS    PARALLEL_DEGREE_LIMIT_ABSOLUTE  Plan/method for apps users  ACTIVE  0
```

Changes to resource plans take immediate effect across all instances.

> **See Also:** *Oracle8i Reference* for details about data dictionary views associated with resource plans and resource consumer groups

Resource Allocation Methods

Resource allocation methods determine what method or policy the Database Resource Manager uses when allocating a particular resource to a resource consumer group or resource plan.

Oracle provides the following methods shown in Table 9–1 for allocating resources to a resource consumer group or resource plan:

Table 9–1 Methods for Allocating Resources

Method	Resource	Resource Recipient
Round-robin method	CPU allocation to sessions	Resource consumer groups
Emphasis method	CPU allocation to consumer groups	Resource plans
Absolute method	Parallel degree limit	Resource plans

CPU Allocation for Resource Plans: Emphasis Method

The *emphasis* CPU allocation method determines how much emphasis is given to sessions in different consumer groups in a resource plan. CPU usage is assigned using levels from 1 to 8, with level 1 having the highest priority. Percentages specify how to allocate CPU to each consumer group at each level.

The following rules apply for the emphasis resource allocation method:

- Sessions in resource consumer groups with non-zero percentages at higher-priority levels always get the first opportunity to run.
- CPU resources are distributed at a given level based on the specified percentages. The percentage of CPU specified for a resource consumer group is a maximum for how much that consumer group can use at a given level. If any

CPU resources are left after all resource consumer groups at a given level have been given an opportunity to run, the remaining CPU resources fall through to the next level. If a consumer group does not consume its allotted resources, then the resources are passed to the next level, not given to the other consumer groups at the same level.

- The sum of percentages at any given level must be less than or equal to 100.
- Any unused CPU time gets recycled. In other words, if no consumer groups are immediately interested in a specific period of CPU time (due to percentages), the consumer groups get another opportunity to use the CPU time, starting at level one.
- Any levels that have no plan directives explicitly specified have a default of 0% for all subplans/consumer groups.

An example of the emphasis method is explained later in this chapter.

The emphasis resource allocation method offers the following advantages:

- Setting percentages enables you to bring CPUs online and offline and to add and remove servers without having to change CPU percentages.
- The amount of CPU resources specified is not proportional to the number of servers, so there is a fine level of control even with a small number of servers.
- Setting percentages avoids the starvation problem associated with priorities. Users do not run at priorities; instead, they run based on the percentages specified for their resource consumer group. In addition, percentages can be used to simulate a priority scheme.

Parallel Degree Limit for Resource Plans: Absolute Method

The *parallel degree limit* allows the administrator to specify a limit on the degree of parallelism of any operation. This parameter is only allowed in directives that refer to resource consumer groups. Currently, the only resource allocation method for a limit on the degree of parallelism is the *absolute* method. *Absolute* refers to the fact that a number (rather than a percentage or fraction, for example) specifies how many processes may be assigned to an operation.

If there are multiple plan directives referring to the same subplan/consumer group, the parallel degree limit for that subplan/consumer group will be the *minimum* of all the incoming values.

> **See Also:** Chapter 23, "Parallel Execution of SQL Statements" for an explanation of parallel execution

Resource Plan Directives

Resource plan directives are a means of:

- Assigning consumer groups or subplans to resource plans
- Allocating resources among consumer groups in the plan by specifying parameters for each resource allocation method

There is one resource plan directive for each entry in the plan.

Examples

This section includes examples of using resource consumer groups, resource plans, resource allocation methods, and resource plan directives.

Using Resource Consumer Groups and Resource Plans

The first step in using the Database Resource Manager is to identify resource requirements using resource consumer groups and resource plans.

Oracle provides a resource plan, SYSTEM_PLAN, which is defined in Table 9–2:

Table 9–2 SYSTEM_PLAN Default Resource Plan

Entry	Level 1	Level 2	Level 3
SYS_GROUP	100%	0%	0%
OTHER_GROUPS	0%	100%	0%
LOW_GROUP	0%	0%	100%

Users SYS and SYSTEM are in the consumer group SYS_GROUP. You can change this. SYSTEM_PLAN gives priority to system sessions. It also specifies a low priority group, LOW_GROUP, which has lower priority than SYS_GROUP and OTHER_GROUPS. It is up to you to decide which user sessions will be part of

LOW_GROUP. You can use this simple Oracle-provided plan if it is appropriate for your environment.

Table 9–3 and Table 9–4 show sample resource plans for BUGDB and MAILDB:

Table 9–3 BUGDB Sample Resource Plan

Entry	Level 1	Level 2	Level 3
Online resource consumer group	80%	0%	0%
Batch resource consumer group	20%	0%	0%
Bug_Maintenance resource consumer group	0%	100%	0%
OTHER_GROUPS	0%	0%	100%

Table 9–4 MAILDB Sample Resource Plan

Entry	Level 1	Level 2	Level 3
Mailusers resource consumer group	0%	80%	0%
Postman resource consumer group	40%	0%	0%
Mail_Maintenance resource consumer group	0%	20%	0%
OTHER_GROUPS	0%	0%	100%

The data in the BUGDB and MAILDB sample resource plans adheres to the emphasis CPU resource allocation method, which enables you to determine the degree of emphasis for sessions in different resource consumer groups by assigning an emphasis percentage for each resource consumer group.

In the MAILDB plan, the Postman resource consumer group is guaranteed 40% of the CPU. The CPU that is left over (60% or more if Postman does not need all of its allotted 40%) is split between the Mailusers resource consumer group and the Mail_Maintenance resource consumer group in a ration of 80:20. If Mailusers and Mail_Maintenance do not use up the remaining 60% of the CPU time, the remaining CPU time goes to OTHER_GROUPS. If there is still CPU time left after OTHER_GROUPS has an opportunity to use it, all of the consumer groups have an opportunity to use the leftover CPU time according to the rules listed on page 9-5.

Using Subplans

A resource plan that is referred to by another plan is called a *subplan*. For example, Table 9–5 is a plan that contains directives for two subplans:

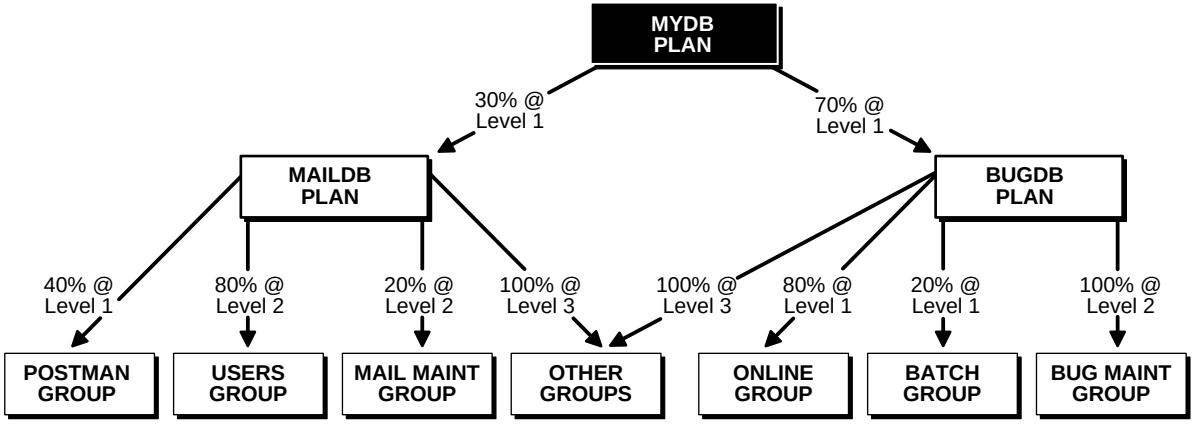
Table 9–5 MYDB Resource Plan, CPU Plan Directives

Subplan/Group	CPU_Level 1
MAILDB Plan	30%
BUGDB Plan	70%

If the MYDB resource plan were in effect and there were an infinite number of runnable users in all resource consumer groups, the MAILDB plan would be in effect 30% of the time, while the BUGDB plan would be in effect 70% of the time.

Breaking this down further, if the MAILDB plan allocates 40% of resources to the Postman resource consumer group and the BUGDB plan allocates 80% of resources to the Online resource consumer group, then users in the Postman group would be run 12% (40% of 30%) of the time, while users in the Online group would be run 56% (80% of 70%) of the time. Figure 9–1 depicts this scenario.

Figure 9–1 Subplans: Resource Plans That Refer to Each Other



Using Multi-Level Resource Plans

A multi-level resource plan is more powerful than a single-level plan. When a resource consumer group does not use its allotment in a level, the remainder falls through to the next level, and you can specify explicitly what to do with it. In the single-level scheme the only choice is to spread the unused time among all the remaining resource consumer groups in the given ratios. Because of this difference, a multi-level scheme cannot be collapsed into a single-level scheme if the percentages for any given level less than the highest level add up to 100.

Using the Parallel Degree Limit Resource Plan Directive

In Table 9–6, the maximum degree of parallelism for any operation issued from the Online Group is 0, 4 for the Batch Group, 4 for the Bug_Maintenance group, and 4 for OTHER_GROUPS. This specification is an example of how the parallel degree limit plan directive can be used to restrict a group of sessions from running parallel operations. Online Group's parallel degree limit is 0. Therefore, all of its operations must run serially.

Table 9–6 Maximum Degree of Parallelism Plan Directives

Subplan/Group	parallel_degree_limit
Online Group	0
Batch Group	4
Bug_Maintenance Group	4
OTHER_GROUPS	4

Summary

Table 9–7 uses the BUGDB plan information from Table 9–3 and combines all the plan directives for the default resource allocation methods:

Table 9–7 Resource Plan Directives

	CPU Resource Plan Directives for Levels 1-8 (Note that only levels that have some non-zero directives need to be specified explicitly)				Parallel Degree Limit Resource Plan Directive
Subplan/Group	**CPU_level 1**	**CPU_level 2**	**CPU_level 3**	**...**	**parallel_degree_limit_1**
ONLINE Group	80%	0%	0%	0%	0
BATCH Group	20%	0%	0%	0%	4
BUG_MT Group	0%	100%	0%	0%	4
OTHER_GROUPS	0%	0%	100%	0%	4

Using the Database Resource Manager

To use the Database Resource Manager, the database administrator:

1. Creates resource plans using the PL/SQL package DBMS_RESOURCE_MANAGER.
2. Creates resource consumer groups using the PL/SQL package DBMS_RESOURCE_MANAGER.
3. Creates resource plan directives using the PL/SQL package DBMS_RESOURCE_MANAGER.
4. Assigns users to consumer groups using the PL/SQL package DBMS_RESOURCE_MANAGER_PRIVS.
5. Specifies the plan to be used by an instance. The initialization parameter RESOURCE_MANAGER_PLAN specifies which top plan to use for a given instance. The Database Resource Manager loads this top plan as well as all its descendants (subplans, directives, and consumer groups).

If the RESOURCE_MANAGER_PLAN parameter is not specified, the Database Resource Manager is disabled. The database administrator can set the parameter dynamically using the ALTER SYSTEM statement to enable the Database Resource

Manager (if it was previously disabled), disable the Database Resource Manager, or change the current plan.

See Also:

- *Oracle8i Supplied PL/SQL Packages Reference*
- *Oracle8i Administrator's Guide*

for information about using these PL/SQL packages

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:

10

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:

- "Database Links" on page 30-9
- "Stored Procedures and Functions" on page 17-2
- Chapter 17, "Procedures and Packages"
- Chapter 19, "Triggers"

for information about additional schema objects

Introduction to Schema Objects

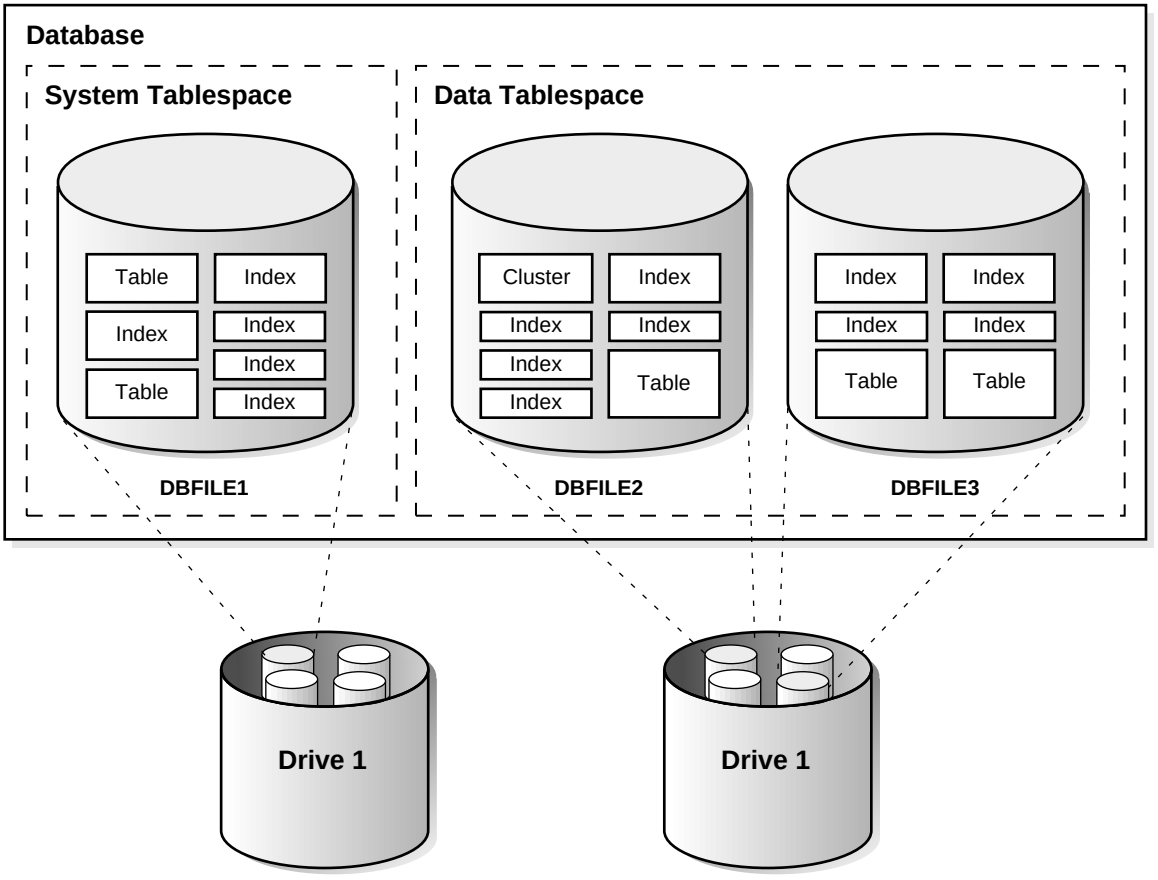
Associated with each database user is a *schema*. A schema is a collection of schema objects. Schema objects include tables, views, sequences, synonyms, indexes, clusters, database links, snapshots, procedures, functions, and packages.

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 10–1 illustrates the relationship among objects, tablespaces, and datafiles.

Figure 10–1 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 might 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.

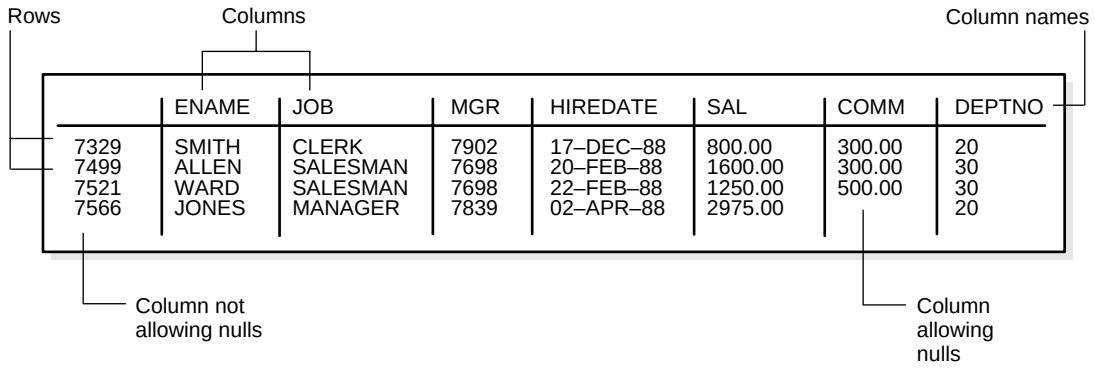
After you create a table, insert rows of data using SQL statements. Table data can then be queried, deleted, or updated using SQL.

Figure 10–2 shows a sample table named EMP.

See Also:

- Chapter 12, "Built-In Datatypes" for a discussion of the Oracle datatypes
- Chapter 25, "Data Integrity" for more information about integrity constraints

Figure 10–2 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.

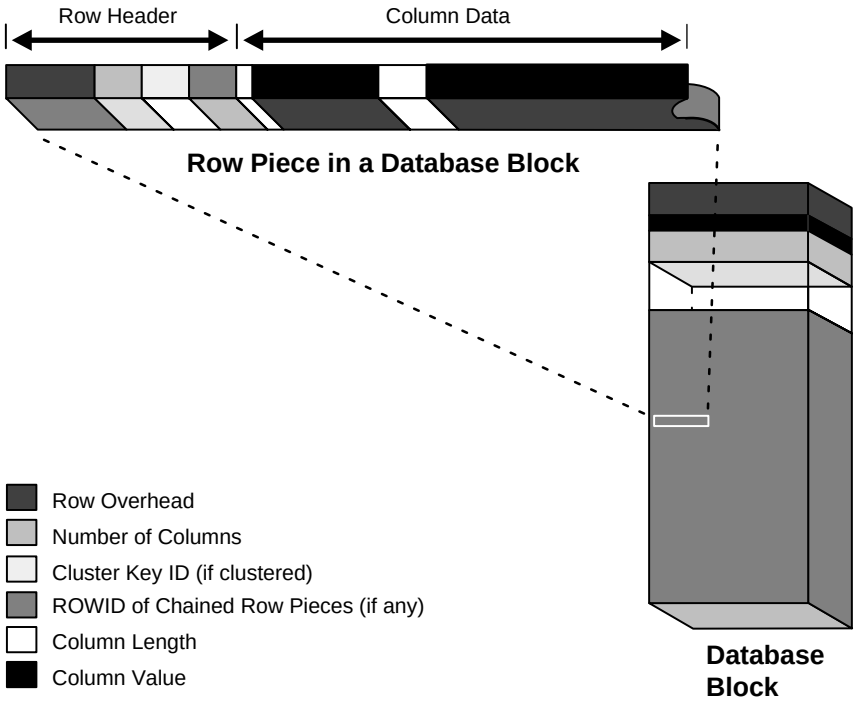
See Also: "User Tablespace Settings and Quotas" on page 26-14

Row Format and Size

Oracle stores each row of a database table 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 per row. When Oracle must store a row in more than one row piece, it is *chained* across multiple blocks. A chained row's pieces are chained together using the rowids of the pieces.

Each row piece, chained or unchained, contains a *row header* and data for all or some of the row's columns. Individual columns might also span row pieces and, consequently, data blocks. Figure 10–3 shows the format of a row piece:

Figure 10–3 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 three bytes of row header. After the row header information, each row contains column length and data. The column length requires one byte for columns that store 250 bytes or less, or three 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 two 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:

- "Row Chaining and Migrating" on page 4-10
- "Nulls" on page 10-8
- "Row Directory" on page 4-4
- "Format of Clustered Data Blocks" on page 10-53

Dropped or Unused Columns

You can drop a column from a table by using the DROP COLUMN clause of the ALTER TABLE statement. This removes the column from the table description and removes the column length and data from each row of the table, freeing space in the data block.

Dropping a column in a large table takes a considerable amount of time. A quicker alternative is to mark a column as unused with the SET UNUSED clause of the ALTER TABLE statement. This makes the column data unavailable, although the data remains in each row of the table. After marking a column as unused, you can add another column that has the same name to the table. The unused column can then be dropped at a later time when you want to reclaim the space occupied by the column data.

See Also:

- *Oracle8i Administrator's Guide* for information about how to drop columns or mark them as unused
- *Oracle8i SQL Reference* for details about the ALTER TABLE statement

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 10-49
- "Physical Rowids" on page 12-16

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, you should 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

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 one 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:

- *Oracle8i SQL Reference* for more information about comparisons using IS NULL and the NVL function
- "Indexes and Nulls" on page 10-25
- "Bitmap Indexes and Nulls" on page 10-38

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, a default value is supplied automatically. Default column values work as though an INSERT statement actually specifies the default value.

Legal default values include any literal or expression that does *not* refer to a column, LEVEL, ROWNUM, or PRIOR. Default values *can* include the SQL functions SYSDATE, USER, USERENV, and UID. The datatype of the default literal or expression must match or be convertible to the column datatype.

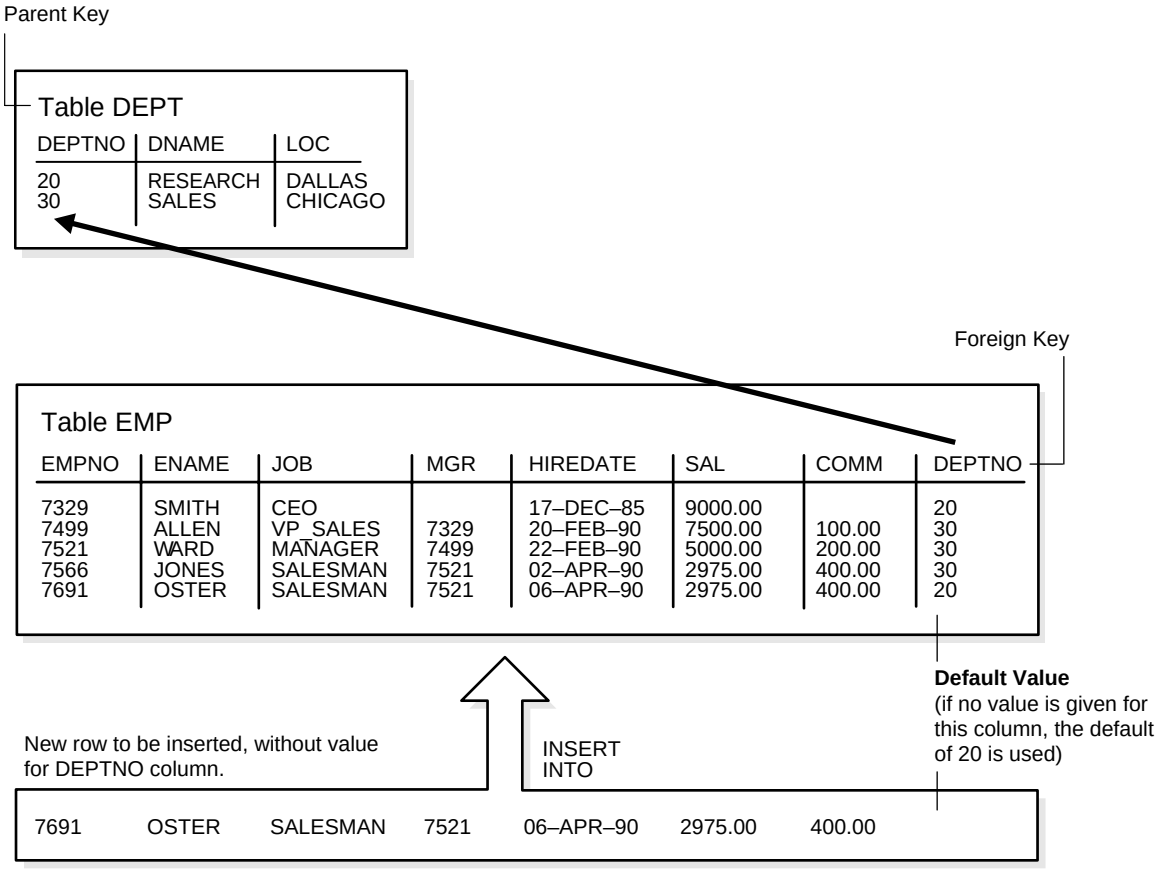
If a default value is not explicitly defined for a column, then the default for the column is implicitly set to NULL.

Default Value Insertion and Integrity Constraint Checking

Integrity constraint checking occurs after the row with a default value is inserted. For example, in Figure 10–4, 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 25, "Data Integrity" for more information about integrity constraints

Figure 10–4 DEFAULT Column Values



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* which 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" on page 13-12
- *Oracle8i 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 which 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 4-17

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 afterwards.

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.

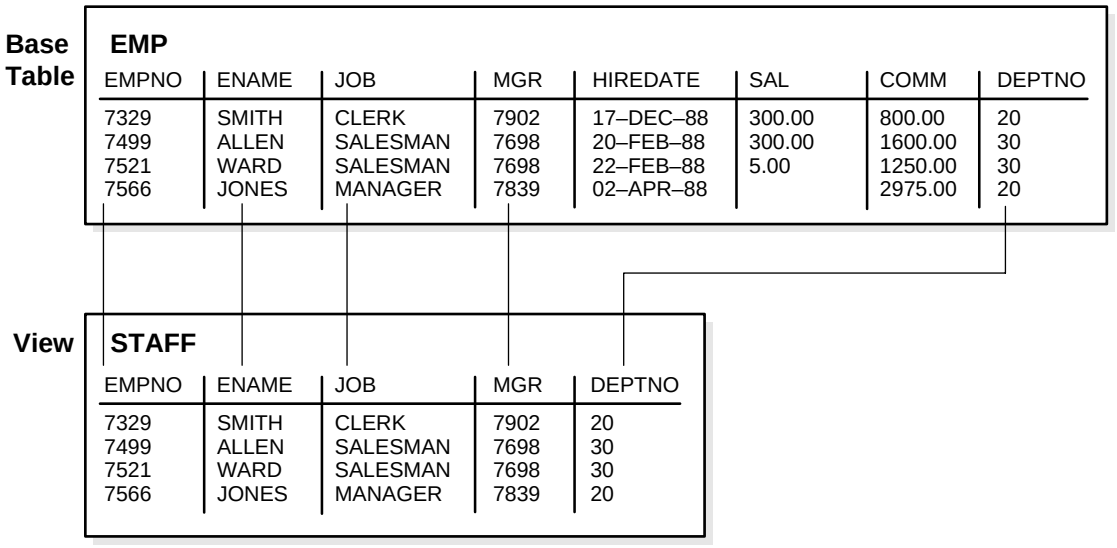
Views

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 10–5 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 10–5 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.

See Also: *Oracle8i SQL Reference*

Note: You cannot explicitly define integrity constraints and triggers on views, but you can define them for the underlying base tables referenced by the view.

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

- To provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table

  For example, Figure 10–5 shows how the STAFF view does not show the SAL or COMM columns of the base table EMP.

- To hide data complexity

  For example, a single view might 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.

- To simplify statements for the user

  For example, views allow users to select information from multiple tables without actually knowing how to perform a join.

- To 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.

- To 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.

- To 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.

- To 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:** *Oracle8i 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.

NLS Parameters

When Oracle evaluates views containing string literals or SQL functions that have NLS parameters as arguments (such as TO_CHAR, TO_DATE, and TO_NUMBER), Oracle takes default values for these parameters from the NLS parameters for the session. You can override these default values by specifying NLS parameters explicitly in the view definition.

> **See Also:** *Oracle8i National Language Support Guide* for information about National Language Support

Using Indexes

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 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:** *Oracle8i Designing and Tuning for Performance* for more information about query optimization

Dependencies and Views

Because a view is defined by a query that references other objects (tables, snapshots, or other views), a view is dependent on the referenced objects. Oracle automatically handles the dependencies for views. For example, if you drop a base table of a view and then recreate it, Oracle determines whether the new base table is acceptable to the existing definition of the view.

> **See Also:** Chapter 20, "Oracle Dependency Management" 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 thatinvolves 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.

Table 10–1 lists rules for updatable join views.

Table 10–1 Rules for INSERT, UPDATE, and DELETE on Join Views

Rule	Description
General Rule	Any INSERT, UPDATE, or DELETE operation on a join view can modify only one underlying base table at a time.
UPDATE Rule	All updatable columns of a join view must map to columns of a key preserved table. If the view is defined with the WITH CHECK OPTION clause, then all join columns and all columns of repeated tables are not updatable.
DELETE Rule	Rows from a join view can be deleted if there is exactly one key-preserved table in the join. If the view is defined with the WITH CHECK OPTION clause and the key preserved table is repeated, then the rows cannot be deleted from the view.
INSERT Rule	An INSERT statement must not explicitly or implicitly refer to the columns of a non-key preserved table. If the join view is defined with the WITH CHECK OPTION clause, then INSERT statements are not permitted.

Views that are not updatable can be modified using INSTEAD OF triggers.

See Also: "INSTEAD OF Triggers" on page 19-13

Object Views

In the Oracle object-relational database, *object views* allow you to retrieve, update, insert, and delete relational data as if they were stored as object types. You can also define views that have columns which are object datatypes, such as objects, REFs, and collections (nested tables and VARRAYs).

See Also:

- Chapter 14, "Object Views"
- *Oracle8i 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: *Oracle8i SQL Reference* for information about subqueries

Materialized Views

Materialized views, also called *snapshots*, 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.

  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 which otherwise would have 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 10-23
- Chapter 11, "Partitioned Tables and Indexes"

Refreshing 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. Another name for materialized view log is *snapshot log*.

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:

- *Oracle8i Data Warehousing Guide* for information about materialized views and materialized view logs in a warehousing environment
- *Oracle8i Replication* for information about materialized views (snapshots) 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, you 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 would all be 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: *Oracle8i Data Warehousing Guide* for information about how dimensions are used in a warehousing environment

The Sequence Generator

The sequence generator provides a sequential series of numbers. The sequence generator is especially useful in multi-user 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.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for performance implications when using sequences

Synonyms

A *synonym* is an alias for any table, view, snapshot, 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 one below:

```
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 an unlimited number of indexes for a table if 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

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 might 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 columns that define the index. Nonunique indexes do not impose this restriction on the column values.

Oracle recommends that you do not explicitly define unique indexes on tables. Uniqueness is strictly a logical concept and should be associated with the definition of a table. Alternatively, define UNIQUE integrity constraints on the desired columns. Oracle enforces UNIQUE integrity constraints by automatically defining a unique index on the unique key.

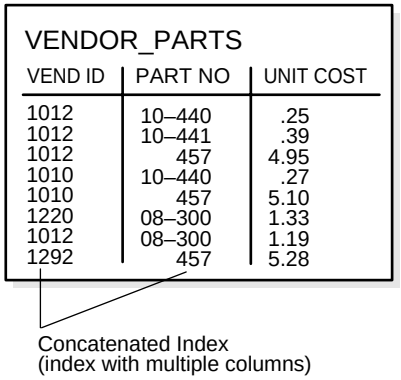
Composite Indexes

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 10–6 illustrates the VENDOR_PARTS table that has a composite index on the VENDOR_ID and PART_NO columns.

Figure 10–6 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: *Oracle8i Designing and Tuning for Performance*

Indexes and Keys

Although the terms are often used interchangeably, you should understand the 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, they should not be confused with each other.

See Also: Chapter 25, "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 10-38

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"
- *Oracle8i Designing and Tuning for Performance*

Uses of Function-Based Indexes

Function-based indexes provide an efficient mechanism for evaluating statements that contain functions in their WHERE clauses. You can create a function-based index to materialize computational-intensive expressions in the index, so that Oracle does not need to compute the value of the expression when it processes SELECT and DELETE statements. 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 an NLS sort index that provides efficient linguistic collation in SQL statements.

> **See Also:** *Oracle8i National Language Support Guide* for information about NLS 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:

- "Statistics for Cost-Based Optimization" on page 21-9 for more information about gathering statistics
- *Oracle8i Designing and Tuning for Performance* for more information about the optimizer

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 will be 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:

- *Oracle8i Designing and Tuning for Performance* for information about DETERMINISTIC PL/SQL functions
- "Function-Based Index Dependencies" on page 20-7 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 26-14

Format of Index Blocks

Space available for index data is the Oracle block size minus block overhead, entry overhead, rowid, and one length byte per 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.

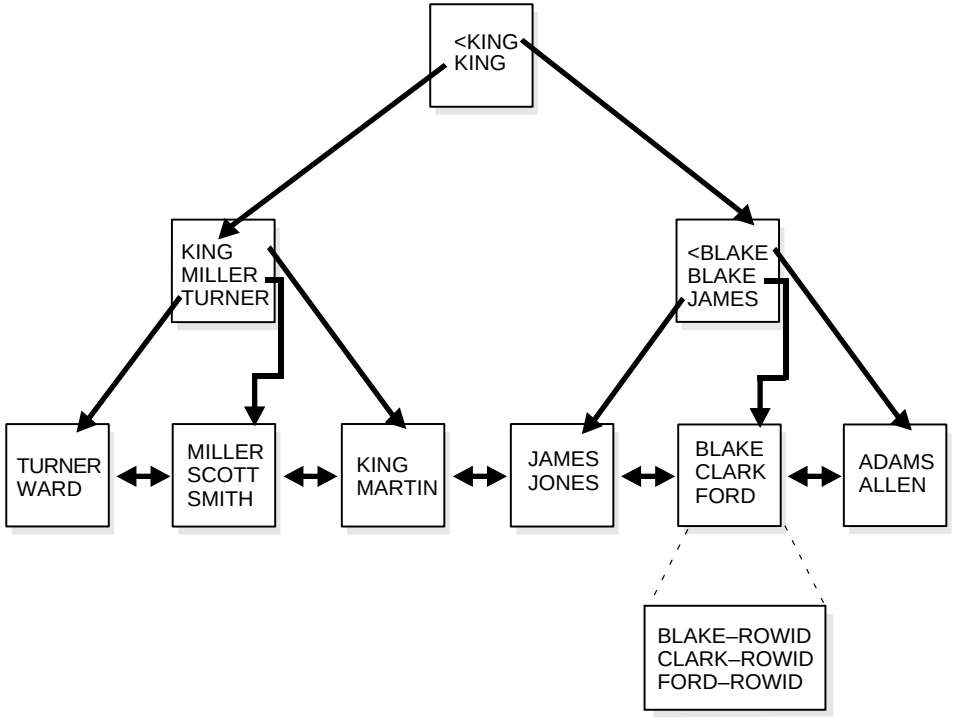
Although Oracle accepts the keywords ASC, DESC, COMPRESS, and NOCOMPRESS in the CREATE INDEX statement, they have no effect on index data, which is stored using rear compression in the branch nodes but not in the leaf nodes.

The Internal Structure of Indexes

Oracle uses B-tree indexes that are balanced to equalize access times to any row. The theory of B-tree indexes is beyond the scope of this manual.

Figure 10–7 illustrates the structure of a B-tree index.

Figure 10–7 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 per 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.

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.

Key Compression

Key compression allows you to 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 non-unique 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, allowing you to store more keys per index block, which can lead to less I/O and better performance.

Although key compression reduces the storage requirements of an index, it may 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 four bytes associated with it.

Uses of Key Compression

Key compression is useful in many different scenarios, such as:

- In a non-unique 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*, *timestamp*), for example (stock_ticker, transaction_time). Thousands of rows might 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 allows you to 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 10-39

Reverse Key Indexes

Creating a *reverse key index*, compared 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 in an Oracle Parallel Server environment 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 Oracle Parallel Server application faster. For example, keeping the index of mail messages in an email 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 Oracle8*i* Enterprise Edition.

See *Getting to Know Oracle8i* for more information about the features available in Oracle8*i* and the Oracle8*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
- Asubstantial 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 since 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 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.

See Also: "Index Partitioning" on page 11-28

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 values in a column are repeated more than a hundred 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 one 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. A regular B-tree index can be several times 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 10–2 shows a portion of a company's customer data.

Table 10–2 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, a unique B-tree index on this column in order would provide the most efficient representation and retrieval.

Table 10–3 illustrates the bitmap index for the REGION column in this example. It consists of three separate bitmaps, one for each region.

Table 10–3 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 bi. 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 might 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 10–8. To identify the specific customers who satisfy the criteria, the resulting bitmap can be used to access the table.

Figure 10–8 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.

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 would only be able to use indexes on columns with NOT NULL constraints.

Example 2

```
SELECT COUNT(*) FROM EMP WHERE COMM IS NULL;
```

This query can be optimized with a bitmap index on COMM.

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 may 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 11, "Partitioned Tables and Indexes" for information about partitioned tables and descriptions of local and global indexes
- *Oracle8i Designing and Tuning for Performance* for more information about using bitmap indexes

Index-Organized Tables

An *index-organized table* differs from an ordinary table because the data for the table is held in its associated index. Changes to the table data, such as adding new rows, updating rows, or deleting rows, result in updating the index.

The index-organized table is like an ordinary table with an index on one or more of its columns, but instead of maintaining two separate storages for the table and the B-tree index, the database system maintains only a single B-tree index which contains both the encoded key value and the associated column values for the corresponding row. Rather than having a row's rowid as the second element of the index entry, the actual data row is stored in the B-tree index. The data rows are built on the primary key for the table, and each B-tree index entry contains `<primary_key_value, non_primary_key_column_values>`.

Index-organized tables are suitable for accessing data by the primary key or any key that is a valid prefix of the primary key. There is no duplication of key values because only non-key column values are stored with the key. You can build secondary indexes to provide efficient access by other columns.

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 10–4 summarizes the differences between index-organized tables and ordinary tables.

Table 10–4 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
Rowid based access	Primary key based access
Sequential scan returns all rows	Full-index scan returns all rows in primary key order
UNIQUE constraint and triggers allowed	UNIQUE constraint not allowed but triggers are allowed
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
Distribution and replication supported	Distribution and replication not supported

See Also:

- "Secondary Indexes on Index-Organized Tables" on page 10-42
- *Oracle8i Administrator's Guide* for information about creating and maintaining index-organized tables

Benefits of Index-Organized Tables

Because data rows are stored in the index, index-organized tables provide faster key-based access to table data for queries that involve exact match or range search or both. The storage requirements are reduced because key columns are not duplicated as they are in an ordinary table and its index. The data row stored with the key in an index-organized table only contains non-key column values. Also, placing the data row with the key eliminates the additional storage that an index on an ordinary table requires for physical rowids, which link the key values to corresponding rows in the table.

Collection Datatypes and Key Compression

Index-organized tables can contain the collection datatypes VARRAY and NESTED TABLE, which store an object ID (OID) for each element of the collection. The storage overhead of 16 bytes per element for the OID is unnecessary, because OIDs are repeated for all elements in a collection. Key compression allows you to compress the repeating OID values in the leaf blocks of an index-organized table

See Also: "Key Compression" on page 10-31

Index-Organized Tables with Row Overflow Area

B-tree index entries are usually quite small since they only consist of the pair `<key, ROWID>`. In index-organized tables, however, the B-tree index entries can be very large since they consist of the pair `<key, non_key_column_values>`. If the index entry gets very large, then the leaf nodes may end up storing one row or row-piece, thereby destroying the dense clustering property of the B-tree index.

Oracle provides an OVERFLOW clause to handle this problem. You can specify an overflow tablespace as well as a threshold value. The threshold is specified as a percentage of the block size (PCTTHRESHOLD).

If the row size is greater than the specified threshold value, then the non-key column values for the row that exceeds the threshold are stored in the specified overflow tablespace. In such a case the index entry contains a `<key, rowhead>` pair, where the rowhead contains the beginning portion of the rest of the columns. It

is like a regular row-piece, except it points to an overflow row-piece that contains the remaining column values.

> **See Also:** *Oracle8i 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 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 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 guesses, access involves two index scans: a secondary index scan followed by a scan of the primary key index.
- With accurate guesses, access involves a secondary index scan and an additional I/O to fetch the data block containing the row.
- With inaccurate guesses, access involves a secondary index scan and an I/O to fetch the wrong data block (as indicated by the guess), followed by a scan of the primary key index.

> **See Also:** "Logical Rowids" on page 12-20

Additional Features of Index-Organized Tables

This section describes some additional features that make index-organized tables more useful.

Rebuilding an Index-Organized Table

You can rebuild an index-organized table to reduce fragmentation from incremental updates. Use the MOVE clause of the ALTER TABLE statement to rebuild an index-organized table.

The MOVE clause rebuilds the primary key index B-tree of the index-organized table but does not rebuild the overflow data segment except when you specify the OVERFLOW clause explicitly or you alter the PCTTHRESHOLD or INCLUDING column value as part of the ALTER TABLE statement. Also, index and data segments associated with LOB columns are not rebuilt unless you specify the LOB column explicitly.

Creating an Index-Organized Table in Parallel

The CREATE TABLE ... AS SELECT statement allows you to create an index-organized table and load it in parallel using the PARALLEL clause in the underlying subquery (AS SELECT). This statement provides an alternative to parallel bulk-load using SQL*Loader.

Partitioning Index-Organized Tables and Their Secondary Indexes

You can partition an index-organized table by range of column values. The partitioning columns must form a subset of the primary key columns.

The following types of secondary indexes on index-organized tables can be partitioned by range of column values:

- Local prefixed index
- Local non-prefixed index
- Global prefixed index

See Also:

- "Range Partitioning" on page 11-13
- "Index Partitioning" on page 11-28

Applications of Interest for Index-Organized Tables

Index-organized tables are especially useful for the following types of applications:

- Information retrieval applications
- Spatial applications

- OLAP applications

Information Retrieval Applications

Information retrieval (IR) applications support content-based searches on document collections. To provide such a capability, IR applications maintain an inverted index for the document collection. An *inverted index* typically contains entries of the form `<token, document_id, occurrence_data>` for each distinct word in a document. The application performs a content-based search by scanning the inverted index looking for tokens of interest.

You can define an ordinary table to model the inverted index. To speed up retrieval of data from the table, you can also define an index on the column corresponding to the token. However, this scheme has the following shortcomings:

- Retrieval of occurrence data from the inverted index using the index incurs an extra rowid-based fetch per row. A typical content-based IR query requires fetching all the inverted index entries for the specified query terms. Because duplicates are the norm rather than the exception in IR applications, a single query term can contain thousands of duplicates. Thus, one rowid-based fetch per row overhead can be very significant, severely impacting the IR search performance.
- Duplication of the key (token) column in the table and in the index leads to wasted storage. Because the inverted index can be huge, storage demands would not be acceptable.

In some cases, retrieval performance can be improved by defining a concatenated index on multiple columns of the inverted index table. The concatenated index allows for index-organized retrieval when the occurrence data is not required, that is, for Boolean queries. In such cases, the rowid fetches of inverted table records is avoided. When the query involves a proximity predicate (for example, the phrase "Oracle Corporation"), the concatenated index approach still requires the inverted index table to be accessed. Furthermore, building and maintaining a concatenated index is much more time-consuming than using a single column index on the token. Also, the storage overhead is higher because multiple columns of the key (token) are duplicated in the table and the index.

Using an index-organized table to model an inverted index overcomes the problems described above. Namely:

- Because the data row is stored along with the key, retrieval of occurrence data for an inverted index involves traversing the index and getting the data rows from the appropriate leaf nodes.

- Only the non-key column values are stored with the key in the index. Thus, there is no duplication of data. Also, this avoids the additional rowid storage overhead which is required if an index is maintained on an ordinary table.

In addition, because index-organized tables are visible to the applications, they are suitable for supporting cooperative indexing where the application and database jointly manage the application-specific indexes.

Spatial Applications

Spatial applications can benefit from index-organized tables as they use some form of inverted index for maintaining application-specific indexes.

Spatial applications maintain inverted indexes for handling spatial queries. For example, a spatial index for objects residing in a collection of grids can be modeled as an inverted index where each entry is of the form:

```
<grid_id, spatial_object_id, spatial_object_data>
```

Index-organized tables are appropriate for modeling such inverted indexes because they provide the required retrieval performance while minimizing storage costs.

OLAP Applications

On-line analytical processing (OLAP) applications typically manipulate multi-dimensional blocks. To allow fast retrieval of portions of the multi-dimensional blocks, they maintain an inverted index to map a set of dimension values to a set of pages.

An entry in the inverted index is of the form:

```
<dimension_value, list_of_pages>
```

The inverted index maintained by OLAP applications can easily be modeled as an index-organized table.

Application Domain Indexes

Oracle provides *extensible indexing* to accommodate indexes on 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:** "User-Defined Datatypes" on page 13-3 for information about object types and their attributes

Indextypes

The *indextype* schema object encapsulates the set of routines that manage and access a domain index. The purpose of an indextype is to enable efficient search and retrieval functions for complex domains such as text, spatial, image, and OLAP data using external application software.

The Oracle Data Cartridge Interface (ODCIIndex) specifies all the routines that must be implemented by the index designer. The routines can be implemented as type methods.

> **See Also:** "Object Types" on page 13-4

Index Definition Routines

The index definition routines:

- Build the domain index when a CREATE INDEX statement references the indextype
- Alter the domain index information when a ALTER INDEX statement alters it
- Remove the index information when a DROP INDEX statement drops it
- Truncate the index when the base table is truncated

Index Maintenance Routines

The index maintenance routines maintain the contents of the domain index when the base table rows are inserted, deleted, updated, or loaded.

Index Scan Routines

The index scan routines implement access to the domain index to retrieve rows of the base table that satisfy predicates containing built-in or user-defined operators in the accessing SQL statement.

An index scan is specified through three routines:

- **istart**, which initializes data structures
- **ifetch**, which fetches rows satisfying the predicate
- **iclose**, which closes the cursor after all rows satisfying the predicate are returned

> **See Also:** "User-Defined Operators" on page 10-48 for more information about operators that can be used in scans of domain indexes

Domain Indexes

The *domain index* schema object is an application-specific index that is created, managed, and accessed by routines supplied by an indextype. It is called a domain index because it indexes data in application-specific domains.

Only single-column domain indexes are currently supported. You can build single-column domain indexes on columns having scalar, object, or LOB datatypes.

You can create multiple domain indexes on the same column only if their indextypes are different. The built-in B-tree index method can be viewed as a distinct indextype in this respect.

Storage of Domain Indexes

A domain index can be stored in an index-organized table or in an external file. The SQL interface for extensible indexing makes no restrictions on the location of the index data except that the application must adhere to the protocol for index definition, maintenance, and search operations.

Metadata for Domain Indexes

For B-tree indexes, you can query the USER_INDEXES view to get index information. To provide similar support for domain indexes, index designers can add any domain-specific metadata in the following manner:

- The index designer can define one or more tables to contain the metadata. The key column of this table must be a unique identifier for the index. This unique

key could be the index name (schema.index). The remainder of the column definitions are at the discretion of the index designer.

- Views can be created that join the system-defined metadata tables with the domain index metadata tables to provide a comprehensive set of information for each instance of a domain index. It is the responsibility of the index designer to provide the view definition.

User-Defined Operators

Oracle provides a set of built-in operators which include arithmetic operators (+, -, *, /), comparison operators (=, >, <), logical operators (NOT, AND, OR), and set operators (UNION). These operators take as input one or more arguments *(operands)* and return a result. They are represented in SQL statements by special characters (+) or keywords (AND). Users and domain cartridge writers can define new operators, which can then be used in SQL statements like built-in operators.

A *user-defined operator* is a schema object identified by a name which could be a character string or a special character or symbol. Like built-in operators, the user-defined operator takes a set of operands as input and returns a result. The implementation of the operator must be provided by the user or domain cartridge writer.

User-defined operators can be invoked anywhere built-in operators can be used, that is, wherever expressions can occur in queries and data manipulation statements, such as:

- The select list of a SELECT statement or subquery
- The condition of a WHERE clause
- The ORDER BY and GROUP BY clauses

For example, if you define a new operator named Contains, which takes as input a text document and a keyword and returns TRUE if the document contains that keyword, you can write a SQL query as:

```
SELECT * FROM employees WHERE Contains(resume, 'Oracle and UNIX');
```

You create an operator by specifying the operator name and its bindings, if any, in a CREATE OPERATOR statement. An operator's *binding* associates it with a user-defined function that provides an implementation for the operator. The binding also identifies the operator with a unique *signature* (the sequence of datatypes of the arguments to the function).

An operator can have multiple bindings as long as they differ in their signatures. Oracle executes the appropriate function when the operator is invoked with a particular signature. An operator created in a schema can be evaluated using functions defined in the same or different schemas.

The user-defined function bound to an operator can be:

- A stand-alone function
- A package function
- An object member method

For example, an operator Contains can be created in the Ordsys schema with two bindings and the corresponding functions that providing the implementation in Text and Spatial domains:

```
CREATE OPERATOR Ordsys.Contains
    BINDING
        (VARCHAR2, VARCHAR2) RETURN BOOLEAN USING text.contains,
        (Spatial.Geo, Spatial.Geo) RETURN BOOLEAN USING Spatial.contains;
```

Although the return datatype is specified as part of operator binding declaration, it does not determine the uniqueness of the binding. However, the specified function must have the same argument and return datatypes as the operator binding.

Operators can also be evaluated using indexes. Oracle uses indexes to efficiently evaluate some built-in operators. For example, a B-tree index can be used to evaluate the comparison operators =, >, and <. Similarly, user-defined domain indexes can be used to efficiently evaluate user-defined operators.

An indextype provides index-based implementation for the operators listed in the indextype definition. The Oracle server can invoke routines specified in the indextype to search the domain index to identify candidate rows and then do further processing, such as filtering, selection, and fetching of rows.

> **See Also:** *Oracle8i Data Cartridge Developer's Guide* for more information about indextypes, domain indexes, and user-defined operators

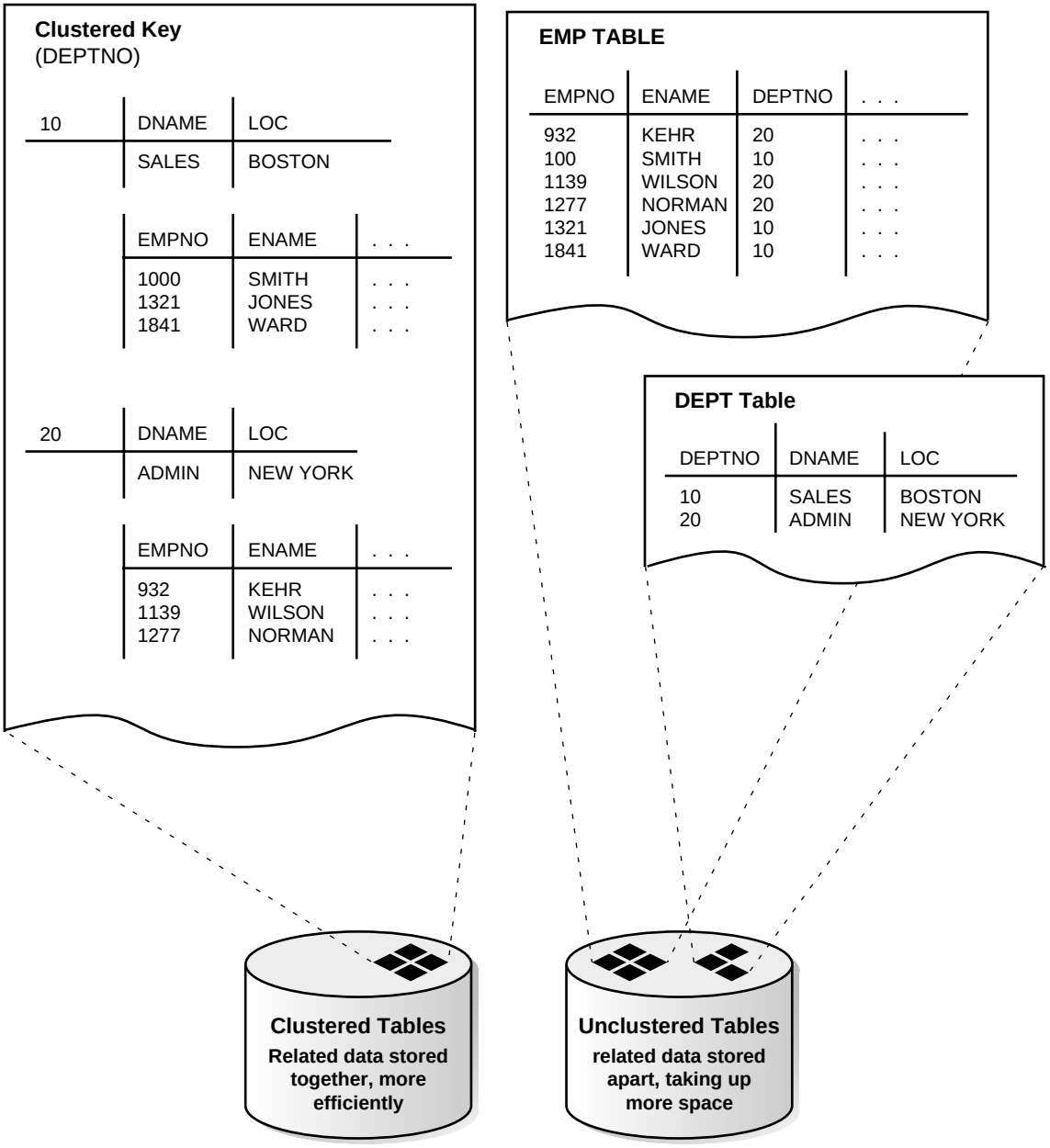
Clusters

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 10–9, "Clustered Table Data" shows what happens when you cluster the EMP and DEPT tables:

Figure 10–9 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 10–9, 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.

Performance Considerations

Clusters can reduce the performance of INSERT statements compared with storing a table separately with its own index. This disadvantage relates to the use of space and the number of blocks that must be visited to scan a table. Because multiple tables have data in each block, more blocks must be used to store a clustered table than if that table were stored nonclustered.

To identify data that would be better stored in clustered form than nonclustered, look for tables that are related by referential integrity constraints and tables that are frequently accessed together using a join. If you cluster tables on the columns used to join table data, you reduce the number of data blocks that must be accessed to process the query because all the rows needed for a join on a cluster key are in the same block. Therefore, performance for joins is improved. Similarly, it might be useful to cluster an individual table. For example, the EMP table could be clustered on the DEPTNO column to cluster the rows for employees in the same department. This would be advantageous if applications commonly process rows department by department.

Like indexes, clusters do not affect application design. The existence of a cluster is transparent to users and to applications. You access data stored in a clustered table via SQL just like data stored in a nonclustered table.

> **See Also:** *Oracle8i Designing and Tuning for Performance*. for more information about the performance implications of using clusters

Format of Clustered Data Blocks

In general, clustered data blocks have an identical format to nonclustered data blocks with the addition of data in the table directory. However, Oracle stores all rows that share the same cluster key value in the same data block.

When you create a cluster, specify the average amount of space required to store all the rows for a cluster key value using the SIZE parameter of the CREATE CLUSTER statement. SIZE determines the maximum number of cluster keys that can be stored per data block.

For example, if each data block has 1700 bytes of available space and the specified cluster key size is 500 bytes, each data block can potentially hold rows for three cluster keys. If SIZE is greater than the amount of available space per data block, each data block holds rows for only one cluster key value.

Although the maximum number of cluster key values per data block is fixed by SIZE, Oracle does not actually reserve space for each cluster key value nor does it guarantee the number of cluster keys that are assigned to a block. For example, if SIZE determines that three cluster key values are allowed per data block, then this does not prevent rows for one cluster key value from taking up all of the available space in the block. If more rows exist for a given key than can fit in a single block, then the block is chained, as necessary.

A cluster key value is stored only once in a data block.

The Cluster Key

The *cluster key* is the column, or group of columns, that the clustered tables have in common. You specify the columns of the cluster key when creating the cluster. You subsequently specify the same columns when creating every table added to the cluster.

For each column specified as part of the cluster key when creating the cluster, every table created in the cluster must have a column that matches the size and type of the column in the cluster key. No more than 16 columns can form the cluster key, and a cluster key value cannot exceed roughly one-half (minus some overhead) the available data space in a data block. The cluster key cannot include a LONG or LONG RAW column.

You can update the data values in clustered columns of a table. However, because the placement of data depends on the cluster key, changing the cluster key for a row might cause Oracle to physically relocate the row. Therefore, columns that are updated often are not good candidates for the cluster key.

The Cluster Index

You must create an index on the cluster key columns after you have created a cluster. A *cluster index* is an index defined specifically for a cluster. Such an index contains an entry for each cluster key value.

To locate a row in a cluster, the cluster index is used to find the cluster key value, which points to the data block associated with that cluster key value. Therefore, Oracle accesses a given row with a minimum of two I/Os—possibly more, depending on the number of levels that must be traversed in the index.

You must create a cluster index before you can execute any DML statements, including INSERT and SELECT statements, against the clustered tables. Therefore, you cannot load data into a clustered table until you create the cluster index.

Like a table index, Oracle stores a cluster index in an index segment. Therefore, you can place a cluster in one tablespace and the cluster index in a different tablespace.

A cluster index is unlike a table index in the following ways:

- Keys that are all null have an entry in the cluster index.
- Index entries point to the first block in the chain for a given cluster key value.
- A cluster index contains one entry per cluster key value, rather than one entry per cluster row.
- The absence of a table index does not affect users, but clustered data cannot be accessed unless there is a cluster index.

If you drop a cluster index, then data in the cluster remains but becomes unavailable until you create a new cluster index. You might want to drop a cluster index to move the cluster index to another tablespace or to change its storage characteristics. However, you must recreate the cluster's index to allow access to data in the cluster.

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.

How Data Is Stored in a Hash Cluster

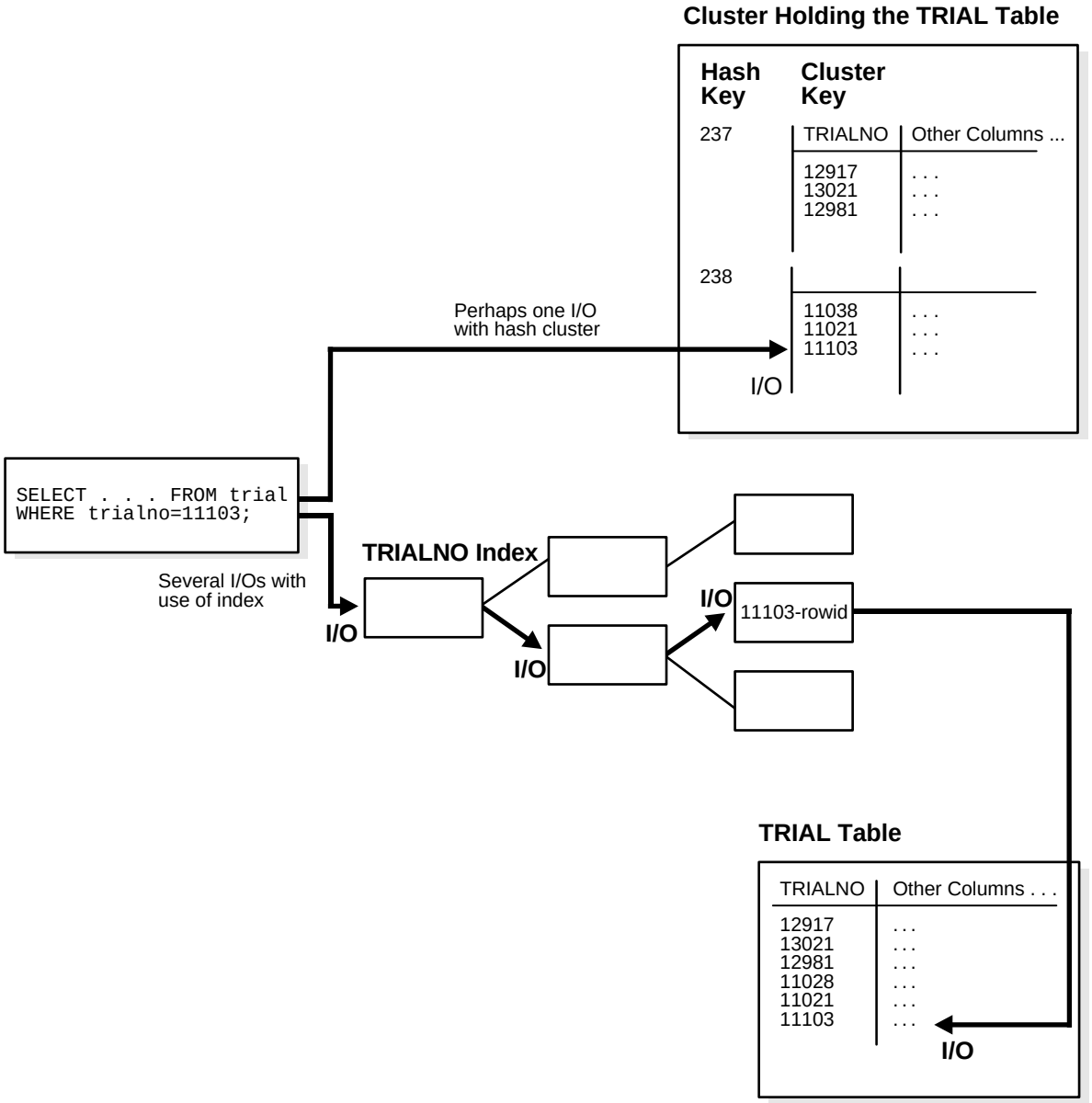
A hash cluster stores related rows together in the same data blocks. Rows in a hash cluster are stored together based on their hash value.

Note: In contrast, an index cluster stores related rows of clustered tables together based on each row's cluster key value.

When you create a hash cluster, Oracle allocates an initial amount of storage for the cluster's data segment. Oracle bases the amount of storage initially allocated for a hash cluster on the predicted number and predicted average size of the hash key's rows in the cluster.

Figure 10–10 illustrates data retrieval for a table in a hash cluster as well as a table with an index. The following sections further explain the internal operations of hash cluster storage.

Figure 10–10 Hashing vs. Indexing: Data Storage and Information



Hash Key Values

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 an issued statement. The number of hash values for a hash cluster is fixed at creation and is determined by the HASHKEYS parameter of the CREATE CLUSTER statement.

The value of HASHKEYS limits the number of unique hash values that can be generated by the hash function used for the cluster. Oracle rounds the number you specify for HASHKEYS to the nearest prime number. For example, setting HASHKEYS to 100 means that for any cluster key value, the hash function generates values between 0 and 100. (There will be 101 hash values.)

Therefore, the distribution of rows in a hash cluster is directly controlled by the value set for the HASHKEYS parameter. With a larger number of hash keys for a given number of rows, the likelihood of a *collision* (two cluster key values having the same hash value) decreases. Minimizing the number of collisions is important because overflow blocks (thus extra I/O) might be necessary to store rows with hash values that collide.

The maximum number of hash keys assigned per data block is determined by the SIZE parameter of the CREATE CLUSTER statement. SIZE is an estimate of the total amount of space in bytes required to store the average number of rows associated with each hash value. For example, if the available free space per data block is 1700 bytes and SIZE is set to 500 bytes, three hash keys are assigned per data block.

> **Note:** The importance of the SIZE parameter of hash clusters is analogous to that of the SIZE parameter for index clusters. However, with index clusters, SIZE applies to rows with the same cluster key value instead of the same hash value.

Although the maximum number of hash key values per data block is determined by SIZE, Oracle does not actually reserve space for each hash key value in the block. For example, if SIZE determines that three hash key values are allowed per block, then this does not prevent rows for one hash key value from taking up all of the available space in the block. If there are more rows for a given hash key value than can fit in a single block, then the block is chained, as necessary.

Note that each row's hash value is not stored as part of the row. However, the cluster key value for each row is stored. Therefore, when determining the proper value for SIZE, the cluster key value must be included for every row to be stored.

See Also: "Single Table Hash Clusters" on page 10-61 for an exception to the rule that Oracle does not reserve space for each hash key value

Hash Functions

A hash function is a function applied to a cluster key value that returns a hash value. Oracle then uses the hash value to locate the row in the proper data block of the hash cluster. The job of a hash function is to provide the maximum distribution of rows among the available hash values of the cluster. To achieve this goal, a hash function must minimize the number of collisions.

Using Oracle's Internal Hash Function

When you create a cluster, you can use the internal hash function of Oracle or bypass the use of this function. The internal hash function allows the cluster key to be a single column or composite key.

Furthermore, the cluster key can consist of columns of any datatype (except LONG and LONG RAW). The internal hash function offers sufficient distribution of cluster key values among available hash keys, producing a minimum number of collisions for any type of cluster key.

Specifying the Cluster Key as the Hash Function

In cases where the cluster key is already a unique identifier that is uniformly distributed over its range, you might want to bypass the internal hash function and simply specify the column on which to hash.

Instead of using the internal hash function to generate a hash value, Oracle checks the cluster key value. If the cluster key value is less than HASHKEYS, then the hash value is the cluster key value. However, if the cluster key value is equal to or greater than HASHKEYS, Oracle divides the cluster key value by the number specified for HASHKEYS, and the remainder is the hash value. That is, the hash value is the cluster key value mod the number of hash keys.

Use the HASH IS parameter of the CREATE CLUSTER statement to specify the cluster key column if cluster key values are distributed evenly throughout the cluster. The cluster key must be comprised of a single column that contains only zero scale numbers (integers). If the internal hash function is bypassed and a non-integer cluster key value is supplied, then the operation (INSERT or UPDATE statement) is rolled back and an error is returned.

Specifying a User-Defined Hash Function

You can also specify any SQL expression as the hash function for a hash cluster. If your cluster key values are not evenly distributed among the cluster, then you should consider creating your own hash function that more efficiently distributes cluster rows more efficiently among the hash values.

For example, if you have a hash cluster containing employee information and the cluster key is the employee's home area code, then it is likely that many employees will hash to the same hash value. To alleviate this problem, you can place the following expression in the HASH IS clause of the CREATE CLUSTER statement:

```
MOD((emp.home_area_code + emp.home_prefix + emp.home_suffix), 101)
```

The expression takes the area code column and adds the phone prefix and suffix columns, divides by the number of hash values (in this case 101), and then uses the remainder as the hash value. The result is cluster rows more evenly distributed among the various hash values.

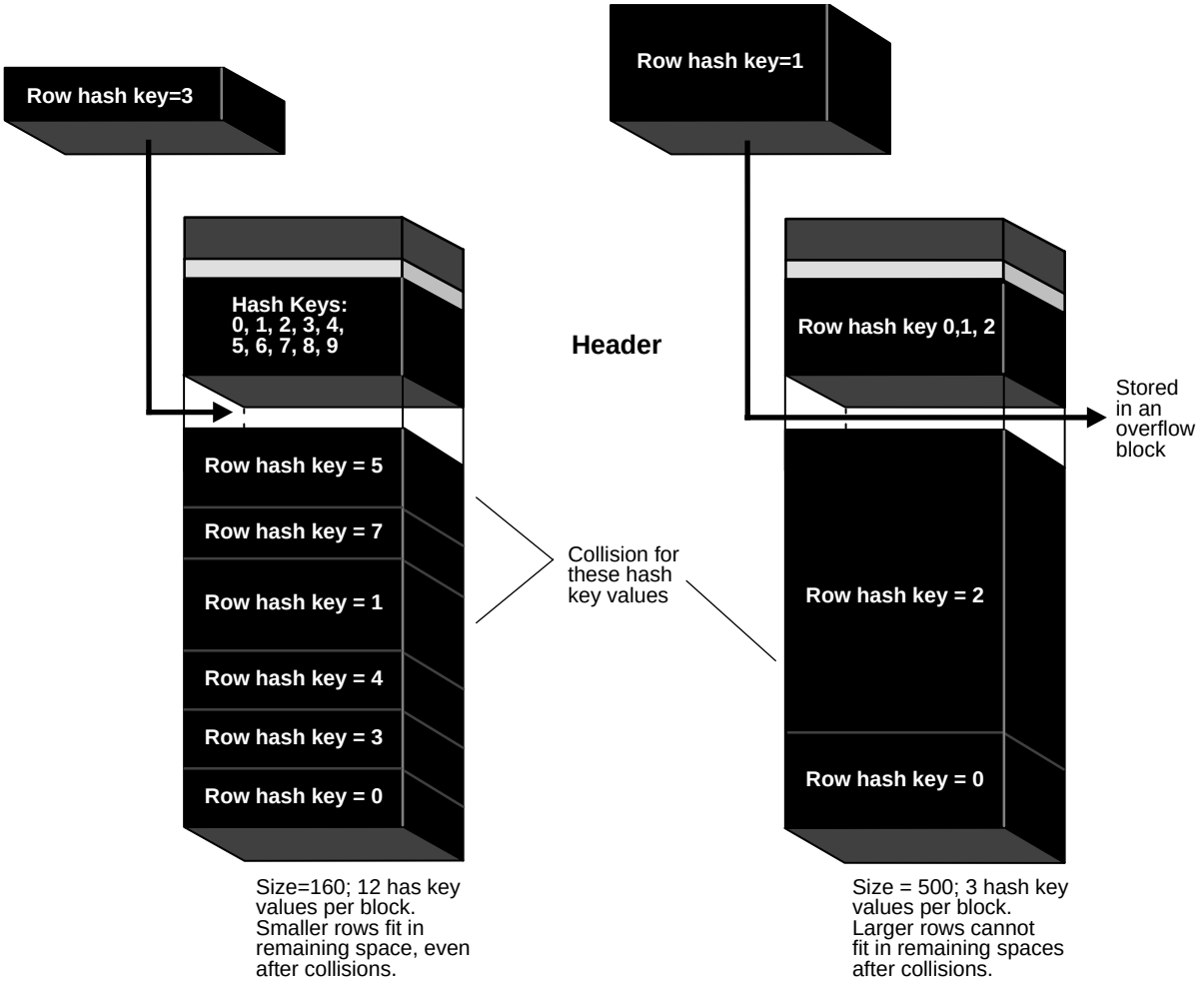
Allocation of Space for a Hash Cluster

As with other types of segments, the allocation of extents during the creation of a hash cluster is controlled by the INITIAL, NEXT, and MINEXTENTS parameters of the STORAGE clause. However, with hash clusters, an initial portion of space, called the *hash table*, is allocated at creation so that all hash keys of the cluster can be mapped, with the total space equal to SIZE * HASHKEYS. Therefore, initial allocation of space for a hash cluster also depends on the values of SIZE and HASHKEYS. The larger of (SIZE*HASHKEYS) and that specified by the STORAGE clause (INITIAL, NEXT, and so on) is used.

Space subsequently allocated to a hash cluster is used to hold the overflow of rows from data blocks that are already full. For example, assume the original data block for a given hash key is full. A user inserts a row into a clustered table such that the row's cluster key hashes to the hash value that is stored in a full data block. Therefore, the row cannot be inserted into the *root block* (original block) allocated for the hash key. Instead, the row is inserted into an overflow block that is chained to the root block of the hash key.

Frequent collisions can result in a larger number of overflow blocks within a hash cluster, thus reducing data retrieval performance. If a collision occurs and there is no space in the original block allocated for the hash key, then an overflow block must be allocated to hold the new row. The likelihood of this happening depends on the average size of each hash key value and corresponding data, specified when the hash cluster is created, as illustrated in Figure 10–11.

Figure 10–11 Collisions and Overflow Blocks in a Hash Cluster



If the average size is small and each row has a unique hash key value, then many hash key values can be assigned per data block. In this case, a small colliding row can likely fit into the space of the root block for the hash key. However, if the average hash key value size is large or each hash key value corresponds to multiple rows, then only a few hash key values can be assigned per data block. In this case, it is likely that the large row will not fit in the root block allocated for the hash key value and an overflow block is allocated.

Single Table Hash Clusters

A single-table hash cluster can provide fast access to rows in a table. In an ordinary hash cluster, Oracle scans all the rows for a given table in the block, even if there actually happens to be just one row with the matching key. In a single-table hash cluster, if there is a one-to-one mapping between hash keys and data row, then Oracle can locate a row without scanning all the rows in the data block.

Oracle preallocates space for each hash key value when the single-table hash cluster is created. There cannot be more than one row per hash value (not the underlying cluster key value), and there cannot be any row chaining in the block. Otherwise Oracle scans all rows in that block to determine which rows match the cluster key.

> **See Also:** *Oracle8i SQL Reference* for details about the SINGLE TABLE HASHKEYS clause of the CREATE CLUSTER statement

11

Partitioned Tables and Indexes

This chapter describes partitioned tables and indexes, and explains some administrative considerations for partitioning. It covers the following topics:

Introduction to Partitioning

This section explains how partitioning can help you manage large tables and indexes in an Oracle database. It includes the following sections:

- What Is Partitioning?
- Advantages of Partitioning
- Manual Partitioning with Partition Views

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.

See Also: *Oracle8i Designing and Tuning for Performance* for information about partitioning materialized views

What Is Partitioning?

Partitioning addresses the key problem of supporting very large tables and indexes by allowing you to decompose them into smaller and more manageable pieces called *partitions*. Once partitions are defined, SQL statements can access and manipulate the partitions rather than entire tables or indexes. Partitions are especially useful in data warehouse applications, which commonly store and analyze large amounts of historical data.

Partitioning Methods

Two primary methods of partitioning are available: r*ange partitioning*, which partitions the data in a table or index according to a range of values, and *hash partitioning*, which partitions the data according to a hash function. Another method, *composite partitioning*, partitions the data by range and further subdivides the data into *subpartitions* using a hash function.

See Also: "Basic Partitioning Model" on page 11-11 for more information about partitioning methods

Logical and Physical Attributes All partitions of a table or index have the same logical attributes, although their physical attributes may be different. For example, all partitions in a table share the same column and constraint definitions, and all

partitions in an index share the same index columns, but storage specifications and other physical attributes such as PCTFREE, PCTUSED, INITRANS, and MAXTRANS may vary for different partitions of the same table or index.

Like partitions, all subpartitions of a table or index have the same logical attributes. Unlike partitions, however, the subpartitions of a single partition cannot have different physical attributes.

Storage of Partitions and Subpartitions A separate segment stores each partition of a range-partitioned or hash-partitioned table or index, and each subpartition of a composite-partitioned table or index. The partitions of a composite-partitioned table or index are logical structures only—they do not occupy separate segments because their data is stored in the segments of their subpartitions.

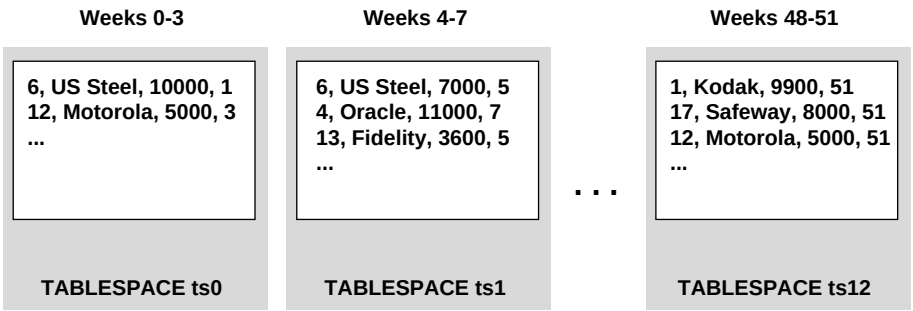
Optionally, you can store each partition (or subpartition of a composite-partitioned table or index) in a separate tablespace, which has the following advantages:

- You can contain the impact of data corruption.
- You can back up and recover each partition or subpartition independently.
- You can map partitions or subpartitions to disk drives to balance the I/O load.

Example of a Partitioned Table

In Figure 11–1, the SALES table contains historical data divided by week number into 13 four-week partitions.

Figure 11–1 SALES Table Partitioned by Week



This SQL statement creates the range-partitioned table shown in Figure 11–1:

```
CREATE TABLE sales ( acct_no        NUMBER(5),
```

```
                acct_name       CHAR(30),
                amount_of_sale  NUMBER(6),
                week_no         INTEGER )
PARTITION BY RANGE ( week_no ) ...
    (PARTITION sales1 VALUES LESS THAN ( 4 ) TABLESPACE ts0,
     PARTITION sales2 VALUES LESS THAN ( 8 ) TABLESPACE ts1,
     ...
     PARTITION sales13 VALUES LESS THAN ( 52 ) TABLESPACE ts12 );
```

See Also: *Oracle8i Administrator's Guide*. for more examples of partitioned tables

Partition Pruning

The Oracle server can explicitly recognize partitions and subpartitions. Oracle uses this knowledge to optimize SQL statements by *pruning* unnecessary partitions or subpartitions. For example, if a query only involves Q1 sales data, there is no need to retrieve data for the remaining three quarters. 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, if the index and table are partitioned on different columns. You can often improve the performance of operations on large tables by creating partitioned indexes which reduce the amount of data that your SQL statements need to access or modify.

The ability to prune unneeded partitions or subpartitions from SQL statements increases performance and availability for many purposes, including:

- Partition-level or subpartition-level load
- Purge
- Backup
- Restore

- Reorganization
- Index building

> **See Also:** "DATE Datatypes" on page 11-21 for more information about what happens if a SQL statement applies a function to a TO_DATE partitioning column

Partition-Wise Joins

An additional area of optimization for partitioned tables is a *partition-wise join*, which is a large join operation that is broken into smaller joins that are performed sequentially or in parallel.

In order to use partition-wise joining, both tables must be equipartitioned.

If the optimizer determines that partition-wise joining provides a performance gain, then it will be used. In some situations, the optimizer can combine pruning and partition-wise joining.

> **See Also:**
>
> - "Equipartitioning" on page 11-23
> - *Oracle8i Designing and Tuning for Performance*

Advantages of Partitioning

This section identifies the classes of databases that could benefit from the use of partitioning, and characterizes them in terms of the problems they present:

- Very Large Databases (VLDBs)
- Reducing Downtime for Scheduled Maintenance
- Reducing Downtime Due to Data Failures
- DSS Performance
- I/O Performance
- Disk Striping: Performance versus Availability
- Partition Transparency

Very Large Databases (VLDBs)

A *very large database (VLDB)* contains hundreds of gigabytes or even a few terabytes of data. Partitioning provides support for VLDBs that contain mostly structured

data, rather than unstructured data. These VLDBs typically owe their size to the presence of a few very large data objects (tables and indexes) rather than to the presence of a very large number of data objects.

There are two major categories of VLDBs:

- *On-Line Transaction Processing (OLTP)* databases are designed for large numbers of concurrent transactions, where each transaction is a relatively simple operation processing a small amount of data.
- *Decision Support Systems (DSS)* are designed for very complex queries that need to access and process large amounts of data.

A VLDB can be characterized as an OLTP database if most of its workload is OLTP. Similarly, a VLDB can be characterized as a DSS database if most of its workload consists of DSS queries.

Partitioning efficiently supports both OLTP VLDBs and DSS VLDBs.

Historical Databases Historical databases are the most common type of DSS VLDB. They contain two classes of tables:

- *Historical* tables describe the business transactions of an enterprise over a recent time interval, such as the last 24 months. There are two types of historical tables:
  - *Base* tables contain baseline information, such as sales, checks, and orders.
  - *Rollup* tables contain summary information derived from the base information using operations such as GROUP BY, AVERAGE, and COUNT.

  The time interval reflected in a historical table is a rolling window, so periodically the database administrator (DBA) deletes the set of rows describing the oldest transactions and allocates space for the set of rows describing new transactions. For example, at the close of business on April 30, 1997 the DBA deletes the rows and all supporting index entries that describe May 1995 transactions and allocates space for May 1997 transactions.

  Most of the data in a historical VLDB is stored in a few very large historical tables that present special problems due to their size and the requirement to smoothly roll out old data and roll in new data.
- *Enterprise* tables describe the business entities of the enterprise, such as departments, locations, and products. This information changes slowly over time and is not modified on a periodic schedule. Although enterprise tables are not large, they affect the performance of many long-running DSS queries that consist of joins of a historical table with enterprise tables.

Partitioning addresses the problem of supporting large historical tables and their indexes by dividing historical data into time-related partitions that can be managed independently and added or deleted conveniently.

Mission-Critical Databases Mission-critical OLTP databases present special availability and performance problems even if they are not very large. For example, it may be necessary to perform scheduled maintenance operations or recover a 10-gigabyte table in a very short period of time, perhaps an hour or less. Also, the DBA may need a degree of control over data placement that is hard to achieve when a table or index is spread over multiple drives.

Partitioning can increase 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. You can also improve access performance to a critical table or index by controlling performance parameters on a partition basis.

Reducing Downtime for Scheduled Maintenance

Partitions enable data management operations like data loads, index creation, and data purges at the partition level, rather than on the entire table, resulting in significantly reduced times for these operations.

Partitioning can significantly reduce the impact of scheduled downtime for maintenance operations:

- By introducing *partition maintenance operations* that operate on an individual partition rather than on an entire table or index
- By providing *partition independence* so that maintenance operations can be performed concurrently on different partitions

> **Note:** Composite-partitioned tables and indexes have *subpartition maintenance operations* as well as partition maintenance operations, and *subpartition independence* as well as partition independence. In the discussion that follows, the general term *partition* refers to both partitions and subpartitions.

Partition Maintenance Operations *Partition maintenance operations* are faster than full table or index maintenance operations. A speedup can be achieved equal to the ratio:

(# records in full table or index) / (# records in partition)

if there are no interpartition stored constructs, such as global indexes and referential integrity constraints.

To further reduce downtime, a partition maintenance operation can take advantage of performance features that are available for table and index-level maintenance operations, such as the PARALLEL, NOLOGGING, and DIRECT (or APPEND) clauses where applicable.

Partition Independence *Partition independence* for the partition maintenance operations makes it possible to perform concurrent maintenance operations on different partitions of the same table or index, as well as concurrent SELECT and DML operations against partitions that are unaffected by maintenance operations.

For example, you can Direct Path Load into partitions PA and PB at the same time, while applications are executing standard SQL SELECT and DML operations against other partitions.

Partition independence is particularly important for operations that involve data movement. Such operations can take a long time (minutes, hours, or even days). Partitioning can reduce the window of unavailability on other partitions to a short time (a few seconds) during operations that involve data movement, if there are no inter-partition stored constructs, such as global indexes and referential integrity constraints.

Partition independence is not needed for short operations with no data movement because these operations complete in a short time.

Reducing Downtime Due to Data Failures

Some maintenance operations are unplanned events, required to recover from hardware or software failures that cause data loss or corruption. Recovery from hardware failures and many system software failures is accomplished by running the RECOVER statement on a database, tablespace, or datafile. Any tables or indexes that have records in a tablespace or datafile being recovered remain unavailable during recovery. Increased availability is particularly important for mission-critical OLTP databases.

Because partitions are independent of each other, the unavailability of a piece or a subset of pieces does not affect access to the rest of the data.

Storing partitions in separate tablespaces provides the following benefits:

- Downtime due to execution of the RECOVER statement is reduced because the unit of recovery (a tablespace) is smaller.

- Disk resources needed for recovery of an offline tablespace (deferred rollback segments) are reduced because the unit of recovery is smaller.
- The amount of unavailable data is reduced, because only the partitions stored in the recovered tablespace have to be taken offline. User applications and maintenance operations can still access the other partitions. This is another example of partition independence.

DSS Performance

DSS queries on very large tables present special performance problems. A query that requires a table scan can take a long time, because it must inspect every row in the table. There is no way to identify and skip subsets of irrelevant rows. The problem is particularly important for historical tables for which many queries concentrate access on rows that were generated recently.

Partitions help solve this DSS performance problem. A query that only requires rows that correspond to a single partition or range of partitions can be executed using a partition scan rather than a table scan.

For example, a query that requests data generated in the month of October 1997 can scan just the rows stored in the October 1997 partition, rather than rows generated over many years of activity. This improves response time and it can also substantially reduce the temporary disk space requirement for queries that require sorts.

I/O Performance

Partitioning can control how data is spread across physical devices. To balance I/O use, you can specify where to store the partitions of a table or index.

With this level of location control, you can accommodate the special needs of applications that require fast response time by reducing disk contention and using faster devices. On the other hand, data that is accessed infrequently, such as old historical data, can be moved to slow disks or stored in subsystems that support a storage hierarchy.

Disk Striping: Performance versus Availability

Disk striping and partitioning are both tools that can improve performance through the reduction of contention for disk arms. Which tool to use, or in which proportions to use them together, is an important issue to consider when physically designing databases. These issues should be considered not only with respect to performance, but also with respect to availability and partition independence.

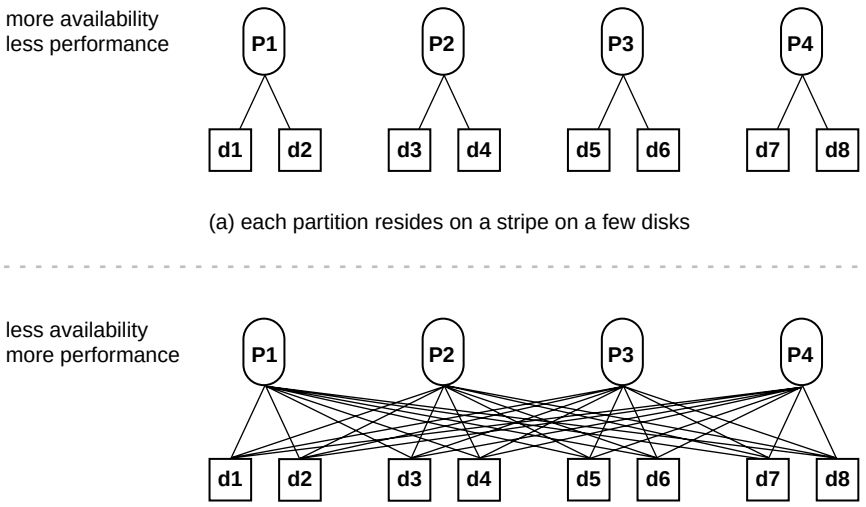
Figure 11–2 shows the two extremes of combining partitioning and striping. Both (a) and (b) in Figure 11–2 show four partitions spread across eight disks, but (a) stripes each partition onto its own pair of disks, whereas (b) stripes each partition onto all eight disks.

The performance characteristics are better in (b), but if any single disk failure occurs, all partitions are adversely affected.

The availability characteristics are better in (a), because failure of a single disk only affects one partition.

Intermediate configurations are also possible, where subsets of partitions are striped over subsets of disks.

Figure 11–2 Partitions and Disk Striping



The trade-off between performance and availability must be decided when determining how to partition tables and indexes, and how to stripe the disks on which they are stored.

For mission-critical databases it is recommended that partition independence and availability be favored. Therefore each partition that you want to stripe across disks should be striped onto its own set of disk drives, which should include enough drives to achieve the required I/O parallelism for accesses to that partition.

Partition Transparency

The vast majority of application programs require *partition transparency*. That is, the programs should be insensitive to whether the data they access is partitioned and how it is partitioned.

A few application programs, however, can take advantage of partitions by explicitly requesting access to an individual partition, rather than the entire table. For example, a user might want to break a long batch job on a very large table into a sequence of short nightly batch jobs on individual partitions.

Manual Partitioning with Partition Views

Instead of using partitioned tables, you can build separate tables with identical templates and define a view that does a UNION of these tables. This is known as *manual partitioning*, and the view is known as a *partition view*. Partition views were the only form of partitioning available in Oracle7. They are not recommended for new applications in Oracle8*i*.

Note: Oracle8*i* supports partition views solely for backwards compatibility with Oracle7. Oracle recommends using partitioned tables instead of partition views.

Partition views that were created for Oracle7 databases can be converted to partitioned tables by using the EXCHANGE PARTITION clause of the ALTER TABLE statement.

> **See Also:** *Oracle8i Administrator's Guide* for instructions on converting partition views to partitioned tables

Basic Partitioning Model

This section describes the basic partitioning model, which includes these partitioning methods:

- Range Partitioning
- Hash Partitioning
- Composite Partitioning

This section also includes these additional topics:

- Partition and Subpartition Names

- Partitioning and Subpartitioning Columns and Keys
- Partition Bounds for Range Partitioning
- Equipartitioning

You can partition a table or index with clauses to the CREATE TABLE or CREATE INDEX statement. After creating a partitioned table or index, you can use an ALTER TABLE or ALTER INDEX statement to modify its partitioning attributes.

The partitioning syntax for CREATE TABLE and CREATE INDEX statements is very similar. The CREATE TABLE statement specifies:

- The logical attributes of the table, such as column and constraint definitions
- The physical attributes of the table

  If the table is nonpartitioned, these are the real physical attributes of the segment associated with the table.

  If the table is partitioned, these table-level attributes specify defaults for the individual partitions of the table.

For a partitioned table, there is also a *partition specification* that includes:

- The table-level algorithm used to map rows to partitions
- A list of partition descriptions, one for each partition in the table
- A list of subpartition descriptions (only for composite partitioning)

Each partition description includes a clause defining supplemental partition-level information about the algorithm used to map rows to partitions. This clause can also specify a partition name and physical attributes for the partition.

Each subpartition description for composite partitioning can specify a subpartition name and a tablespace for the subpartition.

Datatype Restrictions Partitioned tables cannot have any columns with LONG or LONG RAW datatypes. If a table or index is partitioned on a column that has the DATE datatype and if the NLS date format does not specify the century with the year, the partition descriptions must use the TO_DATE function to specify the year completely; otherwise you cannot create the table or index.

> **See Also:** "DATE Datatypes" on page 11-21 for examples of specifying date formats

Bitmap Restrictions 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.

See Also: "Index Partitioning" on page 11-28

Cost Based Optimization Cost based optimization is used when a SQL statement accesses partitioned tables or indexes. Rule base optimization is not available for partitions. A single execution plan is used for all partitions of a partitioned table.

Statistics can be gathered by partition or subpartition, using the DBMS_STATS package or the ANALYZE statement. It is important to gather statistics whenever the nature of the data in a partitioned table changes significantly. The statistics can be found in these data dictionary views:

ALL_TAB_PARTITIONS, DBA_TAB_PARTITIONS, USER_TAB_PARTITIONS

ALL_TAB_SUBPARTITIONS, DBA_TAB_SUBPARTITIONS, USER_TAB_SUBPARTITIONS

ALL_IND_PARTITIONS, DBA_IND_PARTITIONS, USER_IND_PARTITIONS

ALL_IND_SUBPARTITIONS, DBA_IND_SUBPARTITIONS, USER_IND_SUBPARTITIONS

ALL_PART_COL_STATISTICS, DBA_PART_COL_STATISTICS, USER_PART_COL_STATISTICS

ALL_SUBPART_COL_STATISTICS, DBA_SUBPART_COL_STATISTICS, USER_SUBPART_COL_STATISTICS

Range Partitioning

Range partitioning maps rows to partitions based on ranges of column values. Range partitioning is defined by the partitioning specification for a table or index:

```
PARTITION BY RANGE ( column_list )
```

and by the partitioning specifications for each individual partition:

```
VALUES LESS THAN ( value_list )
```

where:

- *column_list* is an ordered list of columns that determines the partition to which a row or an index entry belongs. These columns are called the *partitioning columns*. The values in the partitioning columns of a particular row constitute that row's *partitioning key*.

- *value_list* is an ordered list of values for the columns in *column_list*. Each value in *value_list* must be either a literal or a TO_DATE() or RPAD() function with constant arguments. The *value_list* contained in the partitioning specification for each partition defines an open (noninclusive) upper bound for the partition, referred to as the *partition bound*. The partition bound for each partition must compare less than the partition bound for the next partition.

In each partition, all rows or rows pointed to by index entries have *partitioning keys* that compare less than the *partition bound* for that partition. Unless the partition is the first partition in the table or index, all of its partitioning keys also compare greater than or equal to the partition bound for the previous partition.

For example, in the following table of four partitions (one for each quarter's sales), a row with SALE_YEAR=1997, SALE_MONTH=7, and SALE_DAY=18 has partitioning key (1997, 7, 18). Therefore it belongs in the third partition and is stored in tablespace TSC. A row with SALE_YEAR=1997, SALE_MONTH=7, and SALE_DAY=1 has partitioning key (1997, 7, 1) and also belongs in the third partition, stored in tablespace TSC.

```
CREATE TABLE sales
    ( invoice_no NUMBER,
      sale_year  INT NOT NULL,
      sale_month INT NOT NULL,
      sale_day   INT NOT NULL )
  PARTITION BY RANGE (sale_year, sale_month, sale_day)
    ( PARTITION sales_q1 VALUES LESS THAN (1997, 04, 01)
        TABLESPACE tsa,
      PARTITION sales_q2 VALUES LESS THAN (1997, 07, 01)
        TABLESPACE tsb,
      PARTITION sales_q3 VALUES LESS THAN (1997, 10, 01)
        TABLESPACE tsc,
      PARTITION sales_q4 VALUES LESS THAN (1998, 01, 01)
        TABLESPACE tsd );
```

You can use the ALTER TABLE MERGE PARTITIONS statement to merge the contents of two adjacent range partitions into one partition. You might want to do this to keep historical data online in larger partitions. For example, you might want to have daily partitions, with the oldest partition rolled up into weekly partitions, which can then be rolled up into monthly partitions, and so on.

You can use the ALTER TABLE EXCHANGE PARTITION statement to exchange the data and index segments of:

- A range partition or subpartition with a non-partitioned table

- A non-partitioned table with a range partition or subpartition of a partitioned table

See Also:

- "DATE Datatypes" on page 11-21
- "Partition Bounds for Range Partitioning" on page 11-19 for more information about how partitioning keys are compared to partition bounds

Hash Partitioning

Although partitioning by range is well-suited for historical databases, it may not be the best choice for other purposes. Another method of partitioning, *hash partitioning*, uses a hash function on the partitioning columns to stripe data into partitions. Hash partitioning allows data that does not lend itself to range partitioning to be easily partitioned for performance reasons such as parallel DML, partition pruning, and partition-wise joins.

Hash partitioning is a better choice than range partitioning when:

- You do not know beforehand how much data will map into a given range
- Sizes of range partitions would differ quite substantially
- Partition pruning and partition-wise joins on a partitioning key are important

Hash partitions can be named and stored in specific tablespaces. Local indexes on hash partitions are equipartitioned with the table data. For local index partitions, you can specify the partition names and tablespaces. Storage parameters cannot be specified.

The number of partitions should be a power of two (2, 4, 8, and so on) to obtain the most even data distribution. In addition, cardinality is very important for choosing a key for hash partitioning. For example, suppose that you want to have 16 hash partitions, and that you choose a column with 25 distinct values. Then you might have a scenario where eight of your hash partitions contain a single distinct value, and eight contain two distinct values. Thus, your hash partitions will be widely different sizes. In fact, with such a small number of distinct values, it is even possible that some hash partitions will contain zero distinct values and others will contain three or more distinct values. Since one of the reasons to use hash partitioning is to have evenly sized partitions, Oracle recommends that you do not use hash partitioning on key columns with low cardinality.

In general, the hash-partitioning key should be unique or near unique. Since a table may contain several near-unique keys and thus several candidates for hash partitioning, the other consideration for choosing the partitioning key is the access profile. Hash partitioning can improve the performance of single-key lookups (for example, SELECT * FROM emp WHERE empno = 123), but it does not improve range lookups (for example, SELECT * FROM emp WHERE sal > 5000). Perhaps more importantly, hash partitioning can enable partition-wise joins when two tables are hash-partitioned on the join key. This should be the primary performance consideration for choosing a hash-partitioning key.

The following example creates a table that names and stores a hash partition in a specific tablespace:

```
CREATE TABLE product( … )
    STORAGE (INITIAL 10M)
  PARTITION BY HASH(column_list)
   ( PARTITION p1 TABLESPACE h1,
     PARTITION p2 TABLESPACE h2 );
```

The concepts of splitting, dropping, and merging partitions do not apply to hash partitions. However, you can increase or decrease the number of partitions by using ALTER TABLE to ADD or COALESCE hash partitions.

You can use the ALTER TABLE EXCHANGE PARTITION statement to exchange the data and index segments of:

- A hash partition or subpartition with a non-partitioned table
- A non-partitioned table with a hash partition or subpartition of a partitioned table
- A hash-partitioned table with a hash subpartition of a composite-partitioned table
- A hash subpartition of a composite-partitioned table with a hash-partitioned table

See Also: "Partition Pruning" and "Partition-Wise Joins" on page 11-5

Composite Partitioning

Composite partitioning partitions data using the range method and, within each partition, subpartitions it using the hash method. This type of partitioning supports historical operations data at the partition level and parallelism (parallel DML) and data placement at the subpartition level.

Composite partitioning:

- Provides ease-of-management advantages of range partitioning
- Provides data placement and parallelism advantages of hash partitioning
- Allows you to name the subpartitions and store them in specific tablespaces
- Allows you to build local indexes on composite-partitioned tables, which are stored in the same tablespace as the table subpartition by default
- Allows you to build range-partitioned global indexes
- Allows you to name the index subpartitions and specify their tablespaces

The partitions of a composite-partitioned table or index are logical structures only—their data is stored in the segments of their subpartitions.

The following example creates a table that uses composite partitioning, assuming the NLS DATE format is DD-MON-YYYY:

```
CREATE TABLE orders(
     ordid NUMBER,
     orderdate DATE,
     productid NUMBER,
     quantity NUMBER)
  PARTITION BY RANGE(orderdate)
  SUBPARTITION BY HASH(productid) SUBPARTITIONS 8
  STORE IN(ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8)
   ( PARTITION q1 VALUES LESS THAN('01-APR-1998'),
     PARTITION q2 VALUES LESS THAN('01-JUL-1998'),
     PARTITION q3 VALUES LESS THAN('01-OCT-1998'),
     PARTITION q4 VALUES LESS THAN(MAXVALUE));
```

In this example, the ORDERS table is range partitioned on the ORDERDATE key, in four separate ranges representing quarters of the year. Each range partition is further subpartitioned on the PRODUCTID key into eight subpartitions, for a total of 32 subpartitions. Each tablespace contains one subpartition from each partition.

The following example creates a table that uses composite partitioning with each subpartition explicitly named and stored in a specified tablespace:

```
CREATE TABLE orders( … )
  PARTITION BY RANGE(orderdate)
  SUBPARTITION BY HASH(productid) SUBPARTITIONS 8
  STORE IN (ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8)
   ( PARTITION q1 VALUES LESS THAN('01-APR-1998')
     ( SUBPARTITION q1_h1 TABLESPACE ts1,
```

```
    …
    SUBPARTITION q1_h7 TABLESPACE ts7,
    SUBPARTITION q1_h8 TABLESPACE ts8)
  PARTITION q2 VALUES LESS THAN('01-JUL-1998'), … );
```

Partition and Subpartition Names

Every partition or subpartition has a name, which must conform to the usual rules for naming schema objects and their parts. In particular:

- The name of a table partition or subpartition must be unique among all the partitions or subpartition belonging to the same parent table.
- The name of an index partition or subpartition must be unique among all the partitions or subpartition belonging to the same parent index.

For composite partitioning, the names of subpartitions and partitions are in the same namespace; that is, a partition and a subpartition belonging to the same parent table or index cannot have the same name.

You can rename a partition or subpartition; however, you cannot create any synonyms on a partition or subpartition name.

> **See Also:** *Oracle8i SQL Reference* for more information about the rules for naming schema objects

Referencing a Partition or Subpartition

Partition and subpartition names can optionally be referenced in DDL and DML statements and in utility statements like Import/Export and SQL*Loader. They always appear in context with the name of their parent table or index and they are never qualified by a schema name. The schema name can be used to qualify the parent table or index. For example:

```
ALTER TABLE admin.patient_visits DROP PARTITION pv_dec92;
SELECT * FROM sales PARTITION (s_nov97) s WHERE s.amount_of_sale > 1000;
```

> **See Also:** "Partition-Extended and Subpartition-Extended Table Names" on page 11-63 for more information about referencing partitions and subpartitions in SQL statements

Partitioning and Subpartitioning Columns and Keys

The *partitioning columns* (or *subpartitioning columns*) of a table or index consist of an ordered list of columns whose values determine how the data is partitioned or

subpartitioned. This list can include up to 16 columns, and cannot include any of the following types of columns:

- A LEVEL or ROWID pseudocolumn
- A column of the ROWID datatype
- A nested table, VARRAY, object type, or REF column
- A LOB column (BLOB, CLOB, NCLOB, or BFILE datatype)

A row's *partitioning key* is an ordered list of its values for the partitioning columns. Similarly, in composite partitioning a row's *subpartitioning key* is an ordered list of its values for the subpartitioning columns. Oracle applies either the range or hash method to each row's partitioning key or subpartitioning key to determine which partition or subpartition the row belongs in.

Partition Bounds for Range Partitioning

In a range-partitioned table or index, the partitioning key of each row is compared with a set of upper and lower bounds to determine which partition the row belongs in:

- Every partition of a range-partitioned table or index has a noninclusive *upper bound*, which is specified by the VALUES LESS THAN clause.
- Every partition except the first partition also has an inclusive *lower bound*, which is specified by the VALUES LESS THAN on the next-lower partition.

The partition bounds collectively define an ordering of the partitions in a table or index. The first partition is the partition with the lowest VALUES LESS THAN clause, and the last or highest partition is the partition with the highest VALUES LESS THAN clause.

See Also: "Range Partitioning" on page 11-13

Comparing Partitioning Keys with Partition Bounds

If you attempt to insert a row into a table and the row's partitioning key is greater than or equal to the partition bound for the highest partition in the table, the insert will fail.

When comparing character values in partitioning keys and partition bounds, characters are compared according to their binary values. However, if a character consists of more than one byte, Oracle compares the binary value of each byte, not of the character. The comparison also uses the comparison rules associated with the

column data type. For example, blank-padded comparison is done for the ANSI CHAR data type. The NLS parameters, specifically the initialization parameters NLS_SORT and NLS_LANGUAGE and the environment variable NLS_LANG, have no effect on the comparison.

The binary value of character data varies depending on which character set is being used (for example, ASCII or EBCDIC). For example, ASCII defines the characters A through Z as less than the characters a through z, whereas EBCDIC defines A through Z as being greater than a through z. Thus, partitions designed for one sequence will not work with the other sequence. You must repartition the table after importing from a table using a different character set.

See Also:

- "Multicolumn Partitioning Keys" on page 11-22 for more information about comparing partitioning keys
- *Oracle8i Administrator's Guide* for more information about partition bounds
- *Oracle8i National Language Support Guide* for more information about character set support
- *Oracle8i Utilities* for an example of a table that must be repartitioned after import

MAXVALUE

You can specify the keyword MAXVALUE for any value in the partition bound *value_list*. This keyword represents a virtual infinite value that sorts higher than any other value for the data type, including the NULL value.

For example, you might partition the OFFICE table on STATE (a CHAR(10) column) into three partitions with the following partition bounds:

- VALUES LESS THAN ('I'): States whose names start with A through H
- VALUES LESS THAN ('S'): States whose names start with I through R
- VALUES LESS THAN (MAXVALUE): States whose names start with S through Z, plus special codes for non-U.S. regions

Nulls

NULL cannot be specified as a value in a partition bound *value_list*. An empty string also cannot be specified as a value in a partition bound *value_list*, because it is treated as NULL within the database server.

For the purpose of assigning rows to partitions, Oracle sorts nulls greater than all other values except MAXVALUE. Nulls sort less than MAXVALUE.

This means that if a table is partitioned on a nullable column, and the column is to contain nulls, then the highest partition should have a partition bound of MAXVALUE for that column. Otherwise the rows that contain nulls will map above the highest partition in the table and the insert will fail.

DATE Datatypes

If the partition key includes a column that has the DATE datatype and the NLS date format does not specify the century with the year, you must specify partition bounds using the TO_DATE() function with a 4-character format mask for the year. Otherwise you will not be able to create the table or index.

For example, you might create the SALES table using a DATE column:

```
CREATE TABLE sales
    ( invoice_no NUMBER,
      sale_date DATE NOT NULL )
  PARTITION BY RANGE (sale_date)
    ( PARTITION sales_q1
        VALUES LESS THAN (TO_DATE('1997-04-01','YYYY-MM-DD'))
        TABLESPACE tsa,
      PARTITION sales_q2
        VALUES LESS THAN (TO_DATE('1997-07-01','YYYY-MM-DD'))
        TABLESPACE tsb,
      PARTITION sales_q3
        VALUES LESS THAN (TO_DATE('1997-10-01','YYYY-MM-DD'))
        TABLESPACE tsc,
      PARTITION sales_q4
        VALUES LESS THAN (TO_DATE('1998-01-01','YYYY-MM-DD'))
        TABLESPACE tsd );
```

When you query or modify data, it is recommended that you use the TO_DATE() function in the WHERE clause so that the value of the date information can be determined at compile time. However, the optimizer can prune partitions using a selection criterion on partitioning columns of type DATE when you use another format, as in the following examples:

```
SELECT * FROM sales
  WHERE s_saledate BETWEEN TO_DATE('01-JUL-94', 'DD-MON-YY')
  AND TO_DATE('01-OCT-94', 'DD-MON-YY');

SELECT * FROM sales
```

```
WHERE s_saledate BETWEEN '01-JUL-1994' AND '01-OCT-1994';
```

In this case, the date value will be complete only at runtime. Therefore you will not be able to see which partitions Oracle is accessing as is usually shown on the partition_start and partition_stop columns of the EXPLAIN PLAN statement output on the SQL statement. Instead, you will see the keyword 'KEY' for both columns.

Multicolumn Partitioning Keys

When a table or index is partitioned by range on multiple columns, each partition bound and partitioning key is a list (or vector) of values. The partition bounds and keys are ordered according to ANSI SQL2 vector comparison rules. This is also the way Oracle orders multicolumn index keys.

To compare a partitioning key with a partition bound, you compare the values of their corresponding columns until you find an unequal pair and then that pair determines which vector is greater. The values of any remaining columns have no effect on the comparison.

In mathematical terms, for vectors V1 and V2 which contain the same number of values, Vx[i] is the **i**th value in Vx. Assuming that V1[i] and V2[i] have compatible datatypes:

- V1 = V2 if and only if V1[i] = V2[i] for all i
- V1 < V2 if and only if V1[i] = V2[i] for all i < n and V1[n] < V2[n] for some n
- V1 > V2 if and only if V1[i] = V2[i] for all i < n and V1[n] > V2[n] for some n

For example, if the partition bound for partition P is (7, 5, 10) and the partition bound for the next lower partition is (6, 7, 3) then:

- Key (6, 9, 11) belongs in partition P, because:
  - Key (6, x, x) is less than (7, x, x)
  - Key (6, 9, x) is greater than (6, 7, x)

  Note that the value in the key's third column can be greater than the corresponding value in the partition bound—(x, x, 11) vs. (x, x, 10) in this case—because the comparison does not consider values in the third column after finding an inequality in the second column.
- Key (7, 3, 15) belongs in partition P, because:
  - Key (7, 3, x) is less than (7, 5, x)
  - Key (7, x, x) is greater than (6, x, x)

Note that the value in the key's first column can be equal to the value in the first column of the partition bound. The VALUES LESS THAN clause applies to the columns collectively, not to individual columns.

- Keys (6, 5, 0) and (7, 5, 11) belong in other partitions.

If MAXVALUE appears as an element of a partition bound *value_list*, then the values of all the following elements are irrelevant. For example, a partition bound of (10, MAXVALUE, 5) is equivalent to a partition bound of (10, MAXVALUE, 6) or to a partition bound of (10, MAXVALUE, MAXVALUE).

Multicolumn partitioning keys are useful when the primary key for the table contains multiple columns, but rows are not distributed evenly over the most significant column in the key. For example, suppose that the SUPPLIER_PARTS table contains information about which suppliers provide which parts, and the primary key for the table is (SUPPNUM, PARTNUM). It is not sufficient to partition on SUPPNUM because some suppliers might provide hundreds of thousands of parts, while others provide only a few specialty parts. Instead, you can partition the table on (SUPPNUM, PARTNUM).

Multicolumn partitioning keys are also useful when you represent a date as three CHAR columns instead of a DATE column.

Implicit Constraints Imposed by Partition Bounds

If you specify a partition bound other than MAXVALUE for the highest partition in a table, this imposes an implicit CHECK constraint on the table. This constraint is not recorded in the data dictionary, but the partition bound itself is recorded.

Equipartitioning

Two tables or indexes are *equipartitioned* when the following conditions are true:

- They have the same partitioning method (range or hash), the same partitioning columns, the same number of partitions, and, for range partitioning, the same partition bounds.
- If at least one table or index is composite partitioned, then the tables or indexes are equipartitioned if they are equipartitioned on at least one partitioning method (range or hash). In this case, they are *equipartitioned on one dimension*.

They do not have to be the same type of schema object. For example, a table and an index can be equipartitioned.

Range Equipartitioning

If A and B are range-partitioned tables or indexes, where A[i] is the **i**th partition in A and B[i] is the **i**th partition in B, then A and B are equipartitioned if all of the following are true:

- They have the same number of partitions N.
- They have the same number of partitioning columns M.
- For every 1 <= i <= N, A[i] and B[i] have the same partition bound.

If Apcol[i] is the **i**th partitioning column in A and Bpcol[i] is the **i**th partitioning column in B, then the following must also be true:

- For 1 <= i <= M, Apcol[i] and Bpcol[i] have the same data type, including length, precision, and scale.

A[i] and B[i] may differ in their physical attributes. In particular, they do not have to reside in the same tablespace.

Equipartitioning is important to consider when designing the database:

- It reduces the downtime and the amount of data that is unavailable during partition maintenance operations and tablespace recovery operations. For example, because a table and its local indexes are equipartitioned the effect of splitting a partition is limited to one table partition and the corresponding index partitions. If a table has an index that is not local, then splitting one partition of the table makes it necessary to reorganize the entire index.
- Equipartitioning enables partition-wise join.
- It makes tablespace incomplete recovery (point-in-time recovery) on related subsets of data easier. For example, you might equipartition a table and its primary key index, or a parent table and a child table. You could then recover corresponding partitions to a point in time.

Example of Equipartitioning

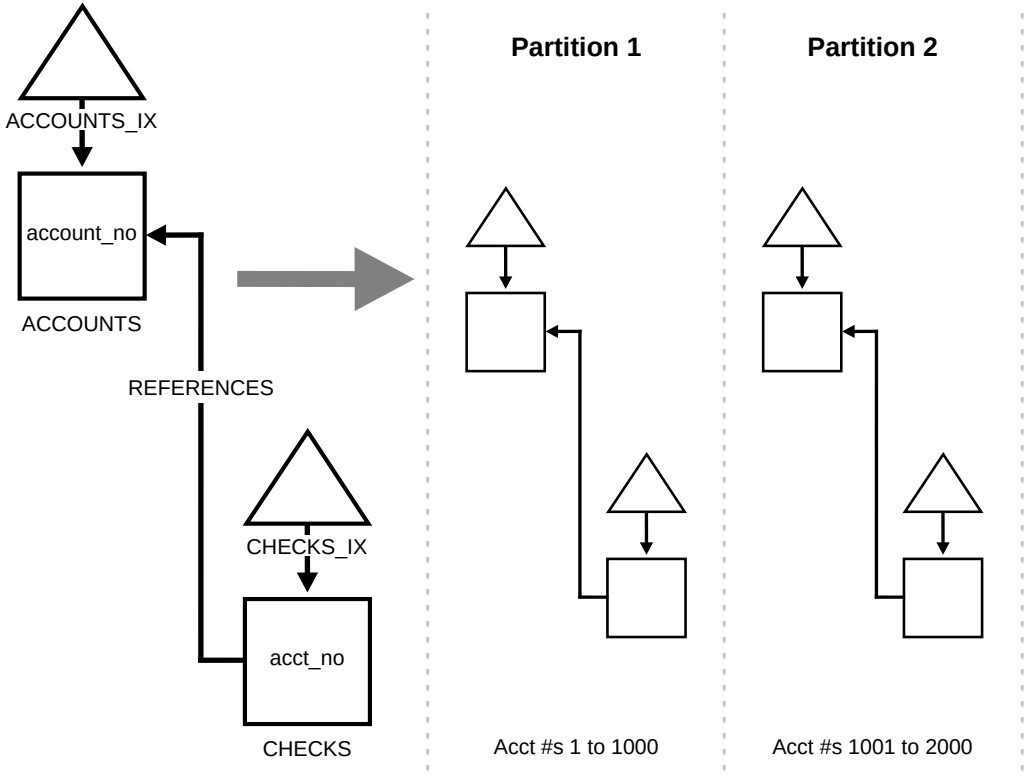
Figure 11–3 shows four logically related schema objects that are equipartitioned:

- ACCOUNTS is a table with two partitions which is range-partitioned on column ACCOUNT_NO. The first partition contains account numbers up to 1000. The second partition contains account numbers up to 2000.
- ACCOUNTS_IX is an index on column ACCOUNT_NO in the ACCOUNTS table. Like the table, the index is range-partitioned on ACCOUNT_NO into two partitions, which have the same partition bounds as partitions of ACCOUNTS.

- CHECKS is a table with two partitions which is range-partitioned on column ACCT_NO. Its partitions have the same partition bounds as partitions of the ACCOUNTS table. ACCT_NO is a foreign key that references ACCOUNT_NO in ACCOUNTS.
- CHECKS_IX is an index on columns (ACCT_NO, CHECK_NO) in CHECKS. It is range-partitioned on ACCT_NO into two partitions, which have the same partition bounds as partitions of ACCOUNTS.

The logical relationship between the four schema objects is shown on the left in Figure 11–3. The physical partitioning is shown on the right. Triangles represent indexes and rectangles represent tables.

Figure 11–3 Equipartitioned Tables and Indexes



Rules for Partitioning Tables and Indexes

This section describes the rules for creating partitioned tables and indexes and the physical attributes of partitions.

Table Partitioning

The rules for partitioning tables are simple:

- A table can be partitioned if:
  - It is not part of a cluster
  - It does not contain LONG or LONG RAW datatypes
- You can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables:
  - A partitioned table can have partitioned and / or nonpartitioned indexes.
  - A nonpartitioned table can have partitioned and / or nonpartitioned indexes. Only global indexes can be created on nonpartitioned tables.

See Also: "Global Partitioned Indexes" on page 11-31

Physical Attributes of Table Partitions

This section discusses the physical attributes of table partitions for range, hash, and composite partitioning.

Range and Hash Partitioning Default physical attributes are initially specified when the CREATE TABLE statement creates a partitioned table. Since there is no segment corresponding to the partitioned table itself, these attributes are only used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using ALTER TABLE MODIFY DEFAULT ATTRIBUTES.

For hash partitioning, all partitions have the same physical characteristics and so the only physical attribute you can specify for a partition is its tablespace.

Physical attributes of table partitions created by CREATE TABLE or ALTER TABLE ADD PARTITION are determined as follows:

- Whenever the value of a partition attribute is not specified, the values of the physical attributes specified (explicitly or by default) for the corresponding base table are used.

For hash partitioning, ALTER TABLE MOVE PARTITION can be used to move the partition to a different tablespace. For range partitioning, this statement can move the partition or modify its physical attributes. Resulting attributes are determined as follows:

- Whenever a new value is not specified, the values that existed before the statement was issued are used.

For range partitioning, the physical attributes of table partitions created by ALTER TABLE SPLIT PARTITION are determined as follows:

- Whenever a new value is not specified, the values of physical attributes of the partition being split are used. This also applies to global index split—missing attributes are inherited from the index partition being split.

Physical attributes of all partitions of a table may be modified by ALTER TABLE, for example, ALTER TABLE *tablename* NOLOGGING changes the logging mode of all partitions of *tablename* to NOLOGGING.

> **See Also:** "Tablespace and Storage Attributes of LOB Data Partitions" on page 11-38 for additional information about the physical attributes of table partitions that contain LOB datatypes

Composite Partitioning For composite partitioning, the partitions specify default physical attributes for the subpartitions and the subpartitions are similar to hash partitions, in that the only physical attribute you can specify explicitly for a subpartition is its tablespace.

The default physical attributes are initially specified when the CREATE TABLE statement creates a composite partitioned table. Since there is no segment corresponding to the partitions or to the table itself, these attributes are only used in derivation of the attributes for member subpartitions. The default attributes can later be modified using ALTER TABLE MODIFY DEFAULT ATTRIBUTES or ALTER TABLE MODIFY DEFAULT ATTRIBUTES FOR PARTITION.

The physical attributes for subpartitions created by CREATE TABLE or ALTER TABLE ADD PARTITION are determined as follows:

- If the tablespace is not specified explicitly for a subpartition, then the tablespace specified (explicitly or by default) for the corresponding partition is used.
- If the values of physical attributes for the partition are not specified, then the attributes specified (explicitly or by default) for the corresponding base table are used.

ALTER TABLE MOVE SUBPARTITION can be used to move a subpartition to a different tablespace, but it does not change other physical attributes of the subpartition. ALTER TABLE MODIFY PARTITION modifies the physical attributes of all of that partition's existing subpartitions as well as the default physical attributes of the partition itself. You can use the FOR PARTITION clause of ALTER TABLE MODIFY PARTITION to avoid changing the attributes of existing subpartitions. Attributes modified at the table level affect the defaults at all three levels: table, partition, and subpartition.

> **See Also:** "Tablespace and Storage Attributes of LOB Data Partitions" on page 11-38 for additional information about the physical attributes of table subpartitions that contain LOB datatypes

Index Partitioning

The rules for partitioning indexes are similar to those for tables:

- An index can be partitioned unless:
  - The index is a cluster index
  - The index is defined on a clustered table.
- You can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables:
  - A partitioned table can have partitioned and/or nonpartitioned indexes.
  - A nonpartitioned table can have partitioned and/or nonpartitioned B-tree indexes.
- Bitmap indexes on nonpartitioned tables cannot be partitioned.
- A bitmap index on a partitioned table must be a local index.

However, partitioned indexes are more complicated than partitioned tables because there are four types of partitioned indexes:

- Local prefixed
- Local nonprefixed
- Global prefixed
- Global nonprefixed

These types are described below. Oracle supports three of the four types. Global nonprefixed indexes are not useful in real applications.

Local Partitioned Indexes

In a *local index*, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is created by specifying the LOCAL attribute.

Oracle constructs the local index so that it is equipartitioned with the underlying table. Oracle partitions the index on the same columns as the underlying table, creates the same number of partitions or subpartitions, and gives them the same partition bounds as corresponding partitions of the underlying table.

Oracle also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced. This ensures that the index remains equipartitioned with the table.

A local index can be created UNIQUE if the partitioning columns form a subset of the index columns. This restriction guarantees that rows with identical index keys always map into the same partition, where uniqueness violations can be detected.

Local indexes have the following advantages:

- Only one index partition needs to be rebuilt when a maintenance operation other than SPLIT PARTITION or ADD PARTITION is performed on an underlying table partition.
- The duration of a partition maintenance operation remains proportional to partition size if the partitioned table has only local indexes.
- Local indexes support partition independence.
- Local indexes support smooth roll-out of old data and roll-in of new data in historical tables.
- Oracle can take advantage of the fact that a local index is equipartitioned with the underlying table to generate better query access plans.
- Local indexes simplify the task of tablespace incomplete recovery. In order to recover a partition or subpartition of a table to a point in time, you must also recover the corresponding index entries to the same point in time. The only way to accomplish this is with a local index. Then you can recover the corresponding table and index partitions or subpartitions together.
- You can build or rebuild local indexes on partitioned tables using intrapartition parallelism (that is, multiple processes for each partition) with the BUILD_PART_INDEX procedure of the DBMS_PCLXUTIL package.

See Also: *Oracle8i Supplied PL/SQL Packages Reference* for a description of the DBMS_PCLXUTIL package

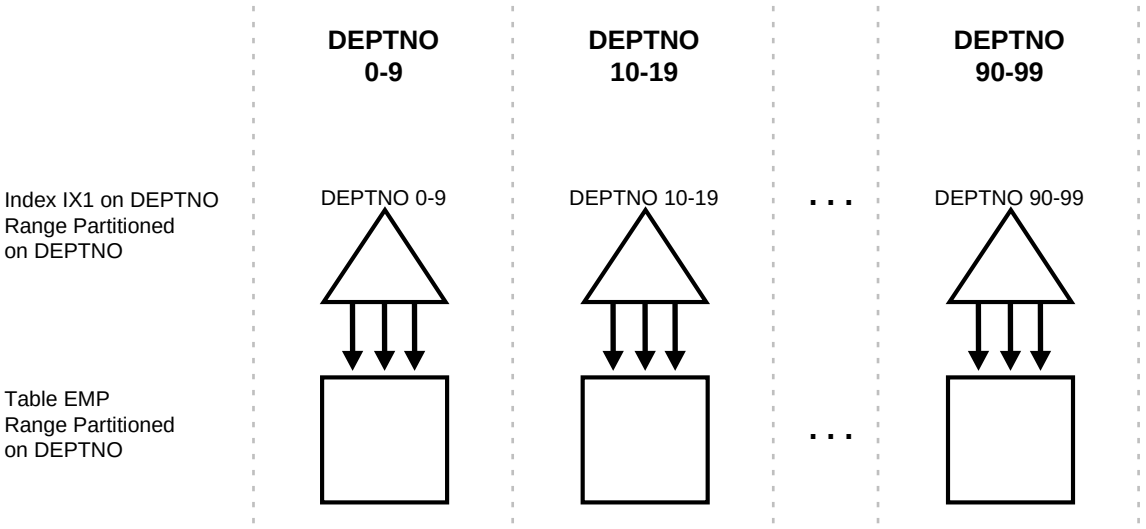
Local Prefixed Indexes A local index is *prefixed* if it is partitioned on a left prefix of the index columns.

For example, if the SALES table and its local index SALES_IX are partitioned on the WEEK_NUM column, then index SALES_IX is *local prefixed* if it is defined on the columns (WEEK_NUM,XACTION_NUM). On the other hand, if index SALES_IX is defined on column PRODUCT_NUM then it is not prefixed.

Figure 11–4 shows another example of a local prefixed index.

Local prefixed indexes can be unique or nonunique.

Figure 11–4 Local Prefixed Index

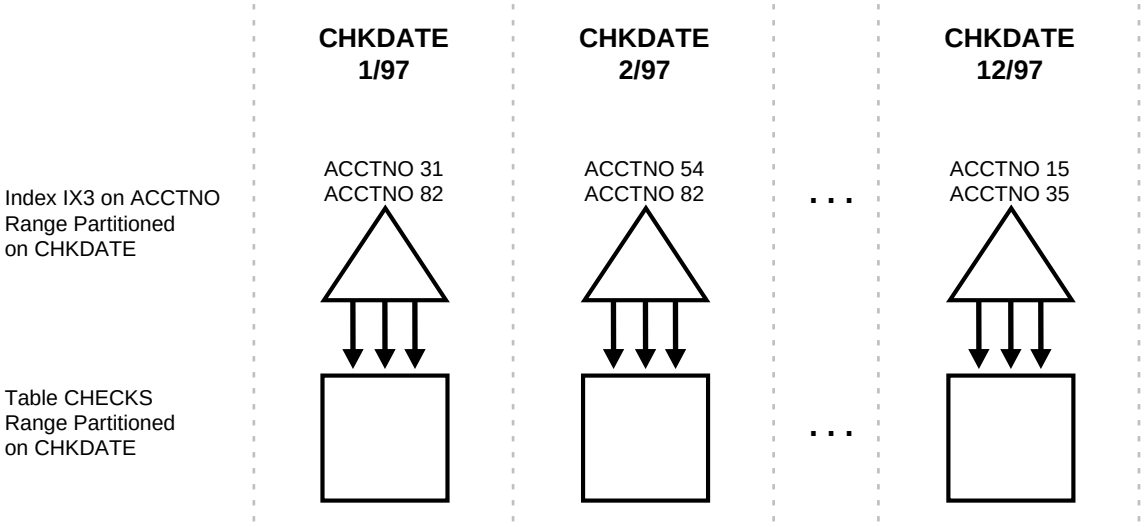


Local Nonprefixed Indexes A local index is *nonprefixed* if it is not partitioned on a left prefix of the index columns.

You cannot have a unique local nonprefixed index unless the partitioning key is a subset of the index key.

Figure 11–5 shows an example of a local nonprefixed index.

Figure 11–5 Local Nonprefixed Index



Global Partitioned Indexes

In a *global partitioned index*, the keys in a particular index partition may refer to rows stored in more than one underlying table partition or subpartition. A global index can only be range-partitioned, but it can be defined on any type of partitioned table.

A global index is created by specifying the GLOBAL attribute. The database administrator is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be merged or split as necessary.

Normally, a global index is not equipartitioned with the underlying table. There is nothing to prevent an index from being equipartitioned with the underlying table, but Oracle does not take advantage of the equipartitioning when generating query plans or executing partition maintenance operations. So an index that is equipartitioned with the underlying table should be created as LOCAL.

A global partitioned index contains a single B-tree with entries for all rows in all partitions. Each index partition may contain keys that refer to many different partitions or subpartitions in the table.

The highest partition of a global index must have a partition bound all of whose values are MAXVALUE. This insures that all rows in the underlying table can be represented in the index.

Prefixed and Nonprefixed Global Partitioned Indexes A global partitioned index is *prefixed* if it is partitioned on a left prefix of the index columns. See Figure 11–6 for an example. A global partitioned index is *nonprefixed* if it is not partitioned on a left prefix of the index columns. Oracle does not support global nonprefixed partitioned indexes.

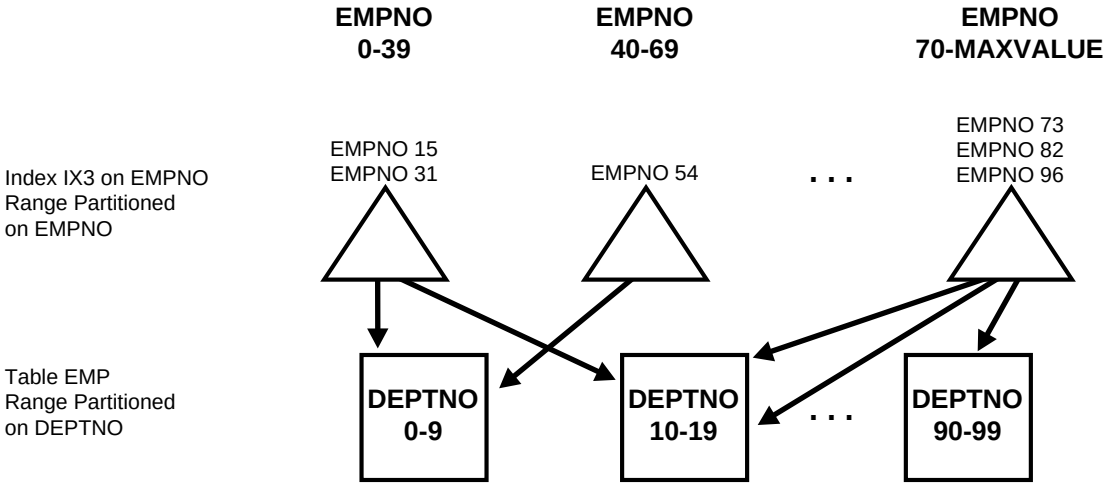
Global prefixed partitioned indexes can be unique or nonunique.

Nonpartitioned indexes are treated as global prefixed nonpartitioned indexes.

Management of Global Partitioned Indexes Global partitioned indexes are harder to manage than local indexes:

- When the data in an underlying table partition is moved or removed (SPLIT, MOVE, DROP, or TRUNCATE), all partitions of a global index are affected. Consequently global indexes cause partition maintenance including rebuilds of global indexes or index partitions to have duration proportional to table size rather than partition size, and they do not support partition independence.
- When an underlying table partition or subpartition is recovered to a point in time, all corresponding entries in a global index must be recovered to the same point in time. Because these entries may be scattered across all partitions or subpartitions of the index, mixed in with entries for other partitions or subpartitions that are not being recovered, there is no way to accomplish this except by re-creating the entire global index.

Figure 11–6 Global Prefixed Partitioned Index



Summary of Partitioned Index Types

Table 11–1 summarizes the types of partitioned indexes that Oracle supports.

- If an index is *local*, it is equipartitioned with the underlying table. Otherwise it is *global*.
- A *prefixed* index is partitioned on a left prefix of the index columns. Otherwise it is *nonprefixed*.

Table 11–1 Types of Partitioned Indexes

Type of Index	Index Equipartitioned with Table	Index Partitioned on Left Prefix of Index Columns	UNIQUE Attribute Allowed	Example		
				Table Partitioning Key	Index Columns	Index Partitioning Key
Local Prefixed (any partitioning method)	Yes	Yes	Yes	A	A,B	A
Local Nonprefixed (any partitioning method)	Yes	No	Yes[1]	A	B	A
Global Prefixed (range partitioning only)	No[2]	Yes	Yes	A	B	B
Global Nonprefixed[3]	—	—	—	—	—	—

1 For a unique local nonprefixed index, the partitioning key must be a subset of the index key.

2 Although a global partitioned index may be equipartitioned with the underlying table, Oracle does not take advantage of the partitioning or maintain equipartitioning after partition maintenance operations such as DROP or SPLIT PARTITION.

3 This type of index is not supported.

Importance of Nonprefixed Indexes

Nonprefixed indexes are particularly useful in historical databases. In a table containing historical data, it is common for an index to be defined on one column to support the requirements of fast access by that column, but partitioned on another column (the same column as the underlying table) to support the time interval for rolling out old data and rolling in new data.

Consider the SALES table presented in Figure 11–1 on page 11-3 ("SALES Table Partitioned by Week"). It contains a year's worth of data, divided into 13 partitions. It is range partitioned on WEEK_NO, four weeks to a partition. You might create a nonprefixed local index SALES_IX on SALES. The SALES_IX index is defined on ACCT_NO because there are queries that need fast access to the data by account number. However, it is partitioned on WEEK_NO to match the SALES table. Every four weeks the oldest partitions of SALES and SALES_IX are dropped and new ones are added.

Performance Implications of Prefixed and Nonprefixed Indexes

It is more expensive to probe into a nonprefixed index than to probe into a prefixed index.

If an index is prefixed (either local or global) and Oracle is presented with a predicate involving the index columns, then partition pruning can restrict application of the predicate to a subset of the index partitions.

For example, in Figure 11–4 on page 11-30 ("Local Prefixed Index"), if the predicate is DEPTNO=15, the optimizer knows to apply the predicate only to the second partition of the index. (If the predicate involves a bind variable, the optimizer will not know exactly which partition but it may still know there is only one partition involved, in which case at run time, only one index partition will be accessed.)

When an index is nonprefixed, Oracle often has to apply a predicate involving the index columns to all N index partitions. This is required to look up a single key, or to do an index range scan. For a range scan, Oracle must also combine information from N index partitions. For example, in Figure 11–5 on page 11-31 ("Local Nonprefixed Index"), a local index is partitioned on CHKDATE with an index key on ACCTNO. If the predicate is ACCTNO=31, Oracle probes all 12 index partitions.

Of course, if there is also a predicate on the partitioning columns, then multiple index probes might not be necessary. Oracle takes advantage of the fact that a local index is equipartitioned with the underlying table to prune partitions based on the partition key. For example, if the predicate in Figure 11–5 is CHKDATE<3/97, Oracle only has to probe two partitions.

So for a nonprefixed index, if the partition key is a part of the WHERE clause but not of the index key, then the optimizer determines which index partitions to probe based on the underlying table partition.

When many queries and DML statements using keys of local, nonprefixed, indexes have to probe all index partitions, this effectively reduces the degree of partition independence provided by such indexes.

Guidelines for Partitioning Indexes

When deciding how to partition indexes on a table, consider the mix of applications that need to access the table. There is a trade-off between performance on the one hand and availability and manageability on the other. Here are some of the guidelines you should consider:

- For OLTP applications:
  - Global indexes and local prefixed indexes provide better performance than local nonprefixed indexes because they minimize the number of index partition probes.

  - Local indexes support more availability when there are partition or subpartition maintenance operations on the table. Local nonprefixed indexes are very useful for historical databases.

- For DSS applications, local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

  For example, a query using the predicate "ACCTNO between 40 and 45" on the table CHECKS of Figure 11–5 on page 11-31 ("Local Nonprefixed Index") causes parallel scans of all the partitions of the nonprefixed index IX3. On the other hand, a query using the predicate "DEPTNO between 40 and 45" on the table DEPTNO of Figure 11–4 on page 11-30 ("Local Prefixed Index") cannot be parallelized because it accesses a single partition of the prefixed index IX1.

- For historical tables, indexes should be local if possible. This limits the impact of regularly scheduled drop partition operations.

- Unique indexes on columns other than the partitioning columns must be global because unique local nonprefixed indexes whose key does not contain the partitioning key are not supported.

Physical Attributes of Index Partitions

Default physical attributes are initially specified when a CREATE INDEX statement creates a partitioned index. Since there is no segment corresponding to the partitioned index itself, these attributes are only used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using ALTER INDEX MODIFY DEFAULT ATTRIBUTES.

Physical attributes of partitions created by CREATE INDEX are determined as follows:

> Values of physical attributes specified (explicitly or by default) for the index are used whenever the value of a corresponding partition attribute is not specified. Handling of the TABLESPACE attribute of partitions of a LOCAL index constitutes an important exception to this rule in that in the absence of a user-specified TABLESPACE value, that of the corresponding partition of the underlying table is used.

Physical attributes (other than TABLESPACE, as explained above) of partitions of local indexes created in the course of processing ALTER TABLE ADD PARTITION are set to the default physical attributes of each index.

Physical attributes (other than TABLESPACE, as explained above) of index partitions created by ALTER TABLE SPLIT PARTITION are determined as follows:

Values of physical attributes of the index partition being split are used.

Physical attributes of an existing index partition can be modified by ALTER INDEX MODIFY PARTITION and ALTER INDEX REBUILD PARTITION. Resulting attributes are determined as follows:

Values of physical attributes of the partition before the statement was issued are used whenever a new value is not specified. Note that ALTER INDEX REBUILD PARTITION can be used to change the tablespace in which a partition resides.

Physical attributes of global index partitions created by ALTER INDEX SPLIT PARTITION are determined as follows:

Values of physical attributes of the partition being split are used whenever a new value is not specified.

Physical attributes of all partitions of an index (along with default values) may be modified by ALTER INDEX, for example, ALTER INDEX *indexname* NOLOGGING changes the logging mode of all partitions of *indexname* to NOLOGGING.

> **See Also:** "Tablespace and Storage Attributes of LOB Index Partitions" on page 11-39 for additional information about the physical attributes of LOB index partitions

Partitioning of Tables with LOB Columns

Tables that contain LOB columns can be partitioned. However, a partitioning key cannot contain a LOB column. The LOB data and LOB index segments of a LOB column are equipartitioned with the base table.

> **Note:** Although this section makes a distinction between the LOB data and LOB index, they are not separate entities. The LOB index, which is implicitly created and maintained by the system, contains control information and is an integral part of LOB column storage.

For every partition of a partitioned table that contains a LOB column, there is a LOB data segment for the LOB data partition and a LOB index segment for the LOB index partition. These data and index segments contain the LOBs that belong to the rows in that partition.

Similarly, for every subpartition of a composite-partitioned table that contains a LOB column, there is a LOB data segment for the LOB data subpartition and a LOB

index segment for the LOB index subpartition. These data and index segments contain the LOBs that belong to the rows in that subpartition. In the following discussion, *partition* refers either to a partition of a range- or hash-partitioned table or index or to a subpartition of a composite-partitioned table or index.

Equipartitioning of LOB data and LOB index segments localizes the effects of maintenance operations, resulting in more efficient use of resources and improved availability of data.

See Also:

- "LOB Datatypes" on page 12-12
- "Partition Maintenance Operations on Tables with LOB Columns" on page 11-56

Tablespace and Storage Attributes of LOB Data Partitions

The algorithm for determining a tablespace for a given LOB data partition is similar to that for determining a tablespace for a LOCAL index partition. In determining values of physical storage attributes other than TABLESPACE for LOB data partitions, Oracle uses the same algorithm that determines the values of physical attributes for table partitions. Note that you can explicitly specify the LOB storage characteristics for a specific LOB column at the partition level or at the table level.

TABLESPACE Attribute of LOB Data Partitions The following rules determine the tablespace of a LOB data partition:

1. If a tablespace is specified for a given LOB data partition, then that value is used.
2. Otherwise, if a default TABLESPACE value, *other than "TABLESPACE DEFAULT"*, is specified at the table level for all LOB data partitions of a given LOB column of the table, then that value is used.
3. Otherwise, the LOB data partition is co-located with the table partition to which it corresponds.

The following example illustrates these rules:

```
CREATE TABLE PT1 (A NUMBER, B BLOB, C CLOB, D CLOB)
   LOB (B,D) STORE AS (STORAGE (NEXT 15K))
   LOB (C) STORE AS (TABLESPACE TSB)
   PARTITION BY RANGE (A)
   (PARTITION P VALUES LESS THAN (MAXVALUE) TABLESPACE TS1
      LOB (B) STORE AS (TABLESPACE TSA),                    (Rule 1)
```

```
    LOB (C) STORE AS (PCTVERSION 20),                    (Rule 2)
    LOB (D) STORE AS (STORAGE (NEXT 10K)))               (Rule 3)
TABLESPACE TSX;
```

In this example, the tablespace for the LOB data partitions corresponding to partition P are determined as follows:

- The LOB data partition for column B is located in tablespace TSA (Rule 1).
- The LOB data partition for column C is located in tablespace TSB (Rule 2).
- The LOB data partition for column D is located in tablespace TS1 (Rule 3)

> **See Also:** "Tablespace and Storage Attributes of LOB Index Partitions" on page 11-39 for a discussion of how to determine a tablespace in which a LOB index partition is located

Other Storage Attributes of LOB Data Partitions The values of the storage attributes (other than TABLESPACE) for a LOB data partition are determined as follows:

1. If a value is specified for a given LOB data partition, then that value is used.
2. Otherwise, if a default value is specified at thethen that value is used.
3. Otherwise, the system or tablespace default value is used. However, in the case of LOGGING, if CACHE is explicitly specified then LOGGING is used regardless of the tablespace value (because CACHE NOLOGGING is not supported).

See the next section for a discussion of how the storage attributes for a LOB index partitions are determined.

Tablespace and Storage Attributes of LOB Index Partitions

LOB index partitions always reside in the same tablespace as the LOB data partitions to which they correspond, that is, the LOB index partitions are co-located with the LOB data partitions. All other attributes of a LOB index partition are determined based on attributes of the LOB data partition to which they correspond and default attributes of the tablespace in which both the LOB data and its corresponding LOB index partition reside.

Note: You cannot specify any attributes for a LOB index or any of its partitions.

TABLESPACE Attribute of LOB Index Partitions The following example shows how LOB index partitions collocate with the LOB data partitions to which they correspond:

```
CREATE TABLE PT1 (A NUMBER, B BLOB, C CLOB, D CLOB)
   LOB (B,D) STORE AS (STORAGE (NEXT 15K))
   LOB (C) STORE AS (TABLESPACE TSB);
   PARTITION BY RANGE (A)
   (PARTITION P VALUES LESS THAN (MAXVALUE) TABLESPACE TS1
      LOB (B) STORE AS (TABLESPACE TSA),                    (Rule 1)
      LOB (C) STORE AS (PCTVERSION 20),                     (Rule 2)
      LOB (D) STORE AS (STORAGE (NEXT 10K)))                (Rule 3)
   TABLESPACE TSX;
```

In this example, the LOB index partitions that correspond to LOB data partitions associated with partition P are located in the following tablespaces:

- The LOB index partition for column B is located in tablespace TSA (Rule 1).
- The LOB index partition for column C is located in tablespace TSB (Rule 2).
- The LOB index partition for column D is located in tablespace TS1 (Rule 3).

Other Storage Attributes of LOB Index Partitions The values of the storage attributes (other than TABLESPACE) for a LOB index partition are determined based on the values of attributes of a corresponding LOB data partition and the default attributes of the tablespace in which the LOB index partition is located.

Views and Partitioned LOBs

Regular views on partitioned tables with LOB columns work the same way that they do on tables without LOB columns. Object views can also be created on top of partitioned tables with LOB columns.

The same view-based privilege checking is performed on LOBs selected from views on partitioned tables as for LOBs selected from views on nonpartitioned tables. The user must have privileges to access the LOB through the view from which the LOB locator is obtained (SELECTed). This view-based privilege checking is necessary for snapshots (that is, materialized views used for replication).

BFILEs in Partitioned Tables

For BFILEs, only the LOB locator is stored in the table while the actual BFILE data exists in an external operating system file. Therefore the BFILE locator is what gets partitioned with the rest of the table, not the BFILE data. The BFILE locator is of varying length and stores the directory alias and file name along with other control

information. Thus, a BFILE column in a partitioned table is similar to a VARCHAR2 column in a partitioned table.

Partitioning Index-Organized Tables and Their Secondary Indexes

You can partition an index-organized table by range of column values. An index-organized table differs from a heap-organized table in the following ways:

- An index-organized table always has a primary key, whereas a regular table may not have a primary key.
- The rows of an index-organized table are stored in the leaf blocks of primary key index segment as part of the index row.
- An index-organized table can optionally have a row overflow data segment in addition to the primary key index segment. Thus, a jeap-organized table without LOBs is stored in a single data segment, whereas an index-organized table without LOBs requires an index segment and optionally an overflow data segment to store the data.

When partitioning an index-organized table, note the following:

- Only range partitioning is supported.
- Partition columns must be a subset of the primary key columns.
- Secondary indexes may also be partitioned, both locally and globally.
- OVERFLOW data segments are always equipartitioned with the table partitions.
- Storage attributes can be specified at the table level or individual partition level for both table data and overflow data.

See Also: "Index-Organized Tables" on page 10-39

Range Partitioning and Primary Key Columns

Restricting the partitioning columns to a subset of the primary key columns ensures that when you insert a row into a partition, the uniqueness of the primary key can be verified by searching that partition. Without this restriction, it would be necessary to search other partitions as well, and so the partitions would not be independent of each other.

When the partitioning columns form a prefix of the primary key columns, the partition bounds form a sequence in primary key order. For queries that require data from more than one partition, a simple concatenation of resulting rows from

each partition preserves the primary key order. This is the optimal way of partitioning an index-organized table.

When the partitioning columns do not form a prefix of the primary key columns, each partition's data is sorted in primary key order but selecting rows from more than one partition in primary key order requires a merge of the individually sorted partition rows.

If you want to partition an index-organized table on columns that are not a subset of the primary key columns, you can use this workaround:

1. Make the partitioning columns part of the primary key by adding them at the end.
2. Define a unique constraint on the original primary key columns.

For example, for an index-organized table that has columns A, B, and C with a primary key (A, B), if you want to partition the table on column C you should change the primary key to (A, B, C) and define a unique constraint on (A, B). Then an insert operation will insert the row into the target partition and insert the key values for (A,B) into a nonpartitioned index on (A,B), which verifies the uniqueness across all the partitions.

Index-Organized Tables without Row Overflow

To create a partitioned index-organized table without row overflow, you need to specify ORGANIZATION INDEX at the table level only. All partitions inherit the ORGANIZATION INDEX property from the table.

You can specify default values for physical attributes at the table level and can override them at the partition level. These attributes apply to the primary key index segment that is created for each partition. The tablespace for an index segment can be specified at the partition level or at the table level. If it is not specified at either level, the user's default tablespace is used.

The following example shows the creation of an index-organized table with no row overflow:

```
CREATE TABLE orders(
   id NUMBER, odate DATE, …
   PRIMARY KEY(id, odate))
   ORGANIZATION INDEX
   PARTITION BY RANGE(odate)
 ( PARTITION p1 … TABLESPACE q1,
   PARTITION p2 … TABLESPACE q2);
```

In this example, the index organized table ORDERS is range partitioned on the ODATE column, with each partition stored in its own tablespace. No overflow is provided for.

Index-Organized Tables with Row Overflow

The overflow option allows storing the tail portion of a row in an overflow data segment. The following are the key aspects for partitioned index-organized tables with overflow:

- For partitioned index-organized tables with overflow, each partition has an index segment and an overflow data segment.
- The overflow data segments are equipartitioned with the primary key index segments.
- Like physical attributes for the index segment, you can specify default values for physical attributes for the overflow data segments at the table level and can override them by specifying partition-level values.
- All the attributes prior to the OVERFLOW keyword apply to the primary key index segment, and all the attributes after the OVERFLOW keyword apply to the overflow data segment.
- The default values for PCTTHRESHOLD and INCLUDING *column* can only be specified at the table level. These clauses control the breaking of the non-key portion into head and tail row-pieces. The head row-piece is stored in the index row and the tail row-piece is stored in the overflow data segment.
- The tablespace for an overflow data segment if not specified for a partition is set to the table-level default. If the table-level default is not specified, then the tablespace of the corresponding partition's index segment is used.
- The system-generated names for the index and overflow data segment are of the form SYS_IOT_TOP_P*n* and SYS_IOT_OVER_P*n*, respectively.

The following example shows the creation of a partitioned index-organized table with partitioned overflow stored in a single tablespace:

```
CREATE TABLE orders(
    id NUMBER, odate DATE, notes VARCHAR2(1000), …
    PRIMARY KEY(id, odate))
    ORGANIZATION INDEX INCLUDING odate
    OVERFLOW TABLESPACE all_overflow
    PARTITION BY RANGE(odate)
     ( PARTITION p1 … TABLESPACE q1,
       PARTITION p2 … TABLESPACE q2);
```

In this example, the table has a separate tablespace for overflow data segments. Even though they are stored in the same physical tablespace (ALL_OVERFLOW), the overflow data segments are partitioned on the same partition columns as used in the index-organized table. Note the use of the INCLUDING ODATE clause—this means that all data, past and including the ODATE column, will be stored in the overflow.

The following example shows the creation of a partitioned index-organized table with partitioned overflow stored in multiple tablespaces:

```
CREATE TABLE orders(
    id NUMBER, odate DATE, notes VARCHAR2(1000), …
    PRIMARY KEY(id, odate))
    ORGANIZATION INDEX INCLUDING odate
    PARTITION BY RANGE(odate)
     ( PARTITION p1 … TABLESPACE q1
       OVERFLOW TABLESPACE q1_overflow,
       PARTITION p2 … TABLESPACE q2
       OVERFLOW TABLESPACE q2_overflow);
```

In this example, each partitioned overflow segment is stored in its own tablespace.

Partitioned Secondary Indexes on Index-Organized Tables

You can create local prefixed, local non-prefixed, and global prefixed partitioned indexes on index-organized tables. Indexes on index-organized tables store primary key-based (logical) rowids as opposed to physical rowids, and may contain additional estimated data as part of the rowid to speed up secondary index-based access.

For accessing an index-organized table partition by its global partition index, Oracle identifies the partition based on the logical rowid. This is possible because the rowid contains primary key columns, which in turn contain all of the partitioning columns. Once the partition is identified, Oracle can use the estimate to directly access the leaf block that would hold the index row. If the estimate is invalid, an index scan on the relevant partition B-tree is required.

See Also:

- "Secondary Indexes on Index-Organized Tables" on page 10-42
- "Logical Rowids" on page 12-20

DML Partition Locks and Subpartition Locks

DML table locks synchronize DML statements (INSERT, UPDATE, and DELETE) with DDL statements and LOCK TABLE statements. DML table locks also synchronize DDL and LOCK TABLE statements among themselves. For partitioned or subpartitioned tables, Oracle uses DML partition locks or DML subpartition locks to provide *partition independence* for DDL and utility operations:

- Range partitioned or hash partitioned tables have *DML partition locks*.
- Tables that use composite partitioning have *DML subpartition locks*.

Partition independence or subpartition independence allows you to perform DDL and utility operations on selected partitions or subpartitions without reducing activity on other partitions or subpartitions.

DML Partition Locks

A partition lock protects the data in an individual partition of a partitioned table while multiple users are accessing that partition or other partitions in the table concurrently.

Partition locks fall between table locks and row locks in the DML locking hierarchy:

- Table locks
  - Partition locks
    - Row locks

Partition locks can be acquired in the same modes as table locks: Share (S), Exclusive (X), Row Share (SS), Row Exclusive (SX), and Share Row Exclusive (SSX).

> **See Also:** "Concurrency Model for Maintenance Operations" on page 11-49 for more information about partition locking for DML and DDL statements

Partition Locking During DML Operations on LOB Columns

When updating a LOB as a whole or only partially (using DBMS_LOB operations), in addition to acquiring a DML SX-lock on a partitioned table, Oracle acquires DML SX-lock(s) on one or more table partitions.

DML Subpartition Locks

DML subpartition locks allow you to perform DDL and utility operations on selected subpartitions without reducing activity on other subpartitions of the same partition, as well as on subpartitions of other partitions.

A subpartition lock protects the data in an individual subpartition while multiple users are accessing that subpartition or other subpartitions in the same partition, or some subpartitions in other partitions of the table concurrently.

Oracle does not acquire DML partition locks when performing DML or DDL operations on composite partitioned tables:

- A DML operation accessing data in a given subpartition acquires a DML lock on that subpartition in the same mode as a similar DML operation accessing data in a partition of a range partitioned table or hash partitioned table.
- A *subpartition maintenance operation* acquires DML locks on subpartitions involved in the operation.
- A maintenance operation on a partition of a composite partitioned table (for example, SPLIT PARTITION or TRUNCATE PARTITION) acquires DML locks on all subpartitions belonging to the partition involved in the operation.

As with partition locks, subpartition locks fall between table locks and row locks in the DML locking hierarchy:

- Table locks
  - Subpartition locks
    - Row locks

Subpartition DML locks can be acquired in the same modes as table and partition DML locks: Share (S), Exclusive (X), Row Share (SS), Row Exclusive (SX), and Share Row Exclusive (SSX).

Performance Considerations for Oracle Parallel Server

Introducing an extra level of DML locking may affect the performance of short transactions in the Oracle Parallel Server environment because extra messages are sent to the Distributed Lock Manager.

To improve performance in the Oracle Parallel Server environment, you can turn off DML locking on selected tables with the ALTER TABLE DISABLE TABLE LOCK statement, which disables both table and partition DML locks. DDL statements are not allowed when DML locking is disabled.

See Also: *Oracle8i Parallel Server Administration, Deployment, and Performance*

Maintenance Operations

This section covers the following topics:

- Partition Maintenance Operations
- Managing Indexes
- Privileges for Partitioned Tables and Indexes
- Auditing for Partitioned Tables and Indexes

For the purposes of this chapter, a *maintenance operation* is a DDL statement that alters a table or index definition or a utility (such as Export, Import, or SQL*Loader) that performs bulk load or unload of data.

Most maintenance operations on nonpartitioned tables and indexes also work on partitioned tables and indexes. For example, DROP TABLE can drop a partitioned table, and Export can export a partitioned table. However, some maintenance operations must be performed on individual partitions rather than the whole partitioned table or index. For example, ALTER TABLE ALLOCATE EXTENT cannot be used for a range-partitioned table; instead, you use ALTER TABLE MODIFY PARTITION ALLOCATE EXTENT for the partition or partitions that need new extents.

Maintenance operations are considered fast if their expected duration is not affected by the size (number of records) of the schema objects they operate upon. Fast maintenance operations result only in dictionary and segment header changes, and do not cause data scans and data updates. They are expected to complete in a short time (order of seconds). For example, RENAME is a fast operation while CREATE INDEX is not a fast operation.

Partition Maintenance Operations

A *partition maintenance operation* modifies one partition of a partitioned table or index. For example, you might add a new partition to an existing table, or you might move a partition to a different tablespace for better I/O load balancing, or you might load a partition.

Some partition maintenance operations are planned events. For example, in a historical database, the database administrator (DBA) periodically drops the oldest partitions from the database and adds a set of new partitions. This drop and add

operation occurs on a regularly scheduled basis. Another example of a planned maintenance operation is a periodic Export/Import to recluster data and reduce fragmentation.

Other partition maintenance operations are unplanned events, required to recover from application or system problems. For example, unexpected transaction activity may force the DBA to split a partition to rebalance I/O load, or the DBA may need to rebuild one or more index partitions.

The partition maintenance operations are:

- Add a table partition or subpartition to an existing table
- Merge table partitions (range or composite partitioning)
- Coalesce table partitions (hash partitioning) or subpartitions (composite partitioning)—redistribute the contents of a partition or subpartition into one or more remaining partitions or subpartitions
- Split an existing partition into two partitions (range or composite partitioning)
- Drop a partition (range or composite partitioning)
- Truncate a table partition or subpartition (with or without reclaiming space)
- Exchange a partition or subpartition—swap the data (and possibly local index segments) of a table partition or subpartition with the data (and index segments) of a non-partitioned table, a hash-partitioned table, or a composite-partitioned table
- Modify a partition or subpartition—change the physical attributes of a partition or subpartition
- Modify default attributes for a partition (composite partitioning)—specify default attributes for new subpartitions of a partition
- Move a table partition or subpartition—move it to another tablespace, or recluster it, or change any of its parameters (including any of its create-time parameters)
- Rename a partition or subpartition
- Mark all local index partitions or subpartitions associated with a table partition or subpartition as UNUSABLE
- Rebuild an index partition or subpartition
- Load data into one table partition or subpartition
- Export data from one table partition or subpartition

- Import a table partition or subpartition

See Also:

- *Oracle8i SQL Reference* for detailed information about partition maintenance operations.
- "Partition Maintenance Operations on Tables with LOB Columns" on page 11-56 for information about maintenance of LOB data

Concurrency Model for Maintenance Operations

The concurrency model described in this section defines when it is possible to run more than one DDL and utility operation on the same schema object at the same time. It also defines which query and DML operations can be run concurrently with DDL and utility operations.

The model applies to all DDL statements. It also applies to utilities like SQL*Loader.

One-Step and Three-Step Operations There are two types of maintenance operations: one-step and three-step.

One-step operations:

- These operations DML lock the affected table in Exclusive (X) mode. Index operations lock the underlying table. They also hold Exclusive dictionary locks for the duration of the operation.
- These operations are either fast (for example, ALTER TABLE ADD PARTITION for range partitioning) or they offer no possibility of running other operations concurrently (for example, ALTER TABLE ADD *column*).
- All index operations are one-step except:
  - CREATE INDEX and ALTER INDEX REBUILD
  - ALTER INDEX DROP or SPLIT PARTITION, if the global partition being dropped or split is USABLE
- All Oracle DDL statements are one-step except:
  - CREATE INDEX
  - MOVE, SPLIT, or REBUILD PARTITION or SUBPARTITION
  - EXCHANGE PARTITION or SUBPARTITION WITH VALIDATION

  - ADD PARTITION for hash partitioning and ADD SUBPARTITION for composite partitioning. This is processed the same way as the SPLIT and MOVE PARTITION or SUBPARTITION operations.
  - COALESCE PARTITION or SUBPARTITION
  - LOAD, EXPORT, or IMPORT PARTITION or SUBPARTITION
- For composite-partitioned tables and local indexes, all partition maintenance operations follow the same protocol as similar operations on range-partitioned tables and local indexes.

Three-step operations:

- These operations acquire less restrictive DML locks on the affected table. They lock only one partition or subpartition in Exclusive (X) mode, or if they lock the entire table, they lock it in an S or SS or SX mode.
- These operations consist of three steps:
  - Step 1: read dictionary while holding Share dictionary locks. Step 1 takes a short time (seconds). At the end of this step, the appropriate DML locks are acquired, then the dictionary locks are released.
  - Step 2: scan or update table or index records. Step 2 may take a long time (minutes or hours).
  - Step 3: update dictionary while holding Exclusive dictionary locks. Step 3 takes a short time (seconds).
- These operations are long running, but they allow other operations to run concurrently. Exactly which operations can run concurrently depends on the specific DML locks acquired by the statement, as explained below.
- If a three-step operation on range-partitioned tables or local indexes acquires DML locks on partitions, the same operation on composite-partitioned tables or local indexes also acquires locks on all subpartitions of each partition involved in the operation.
- The following operations are three-step:
  - ALTER TABLE MOVE PARTITION or SUBPARTITION, ALTER TABLE SPLIT PARTITION, ALTER TABLE EXCHANGE PARTITION or SUBPARTITION WITH VALIDATION, Direct Path Load Table Partition or Subpartition: These statements lock the table in Row Exclusive (SX) mode and the partition or subpartition in Exclusive (X) mode.

- CREATE INDEX and ALTER INDEX REBUILD PARTITION (for a global index): These statements lock the table in Shared (S) mode.
- ALTER INDEX REBUILD PARTITION or SUBPARTITION (for a local index): This statement locks the table in Row Share (SS) mode and the partition or subpartition in Shared (S) mode.
- ALTER TABLE MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES: This statement locks the table in Row Share (SS) mode and the partition in Shared (S) mode.
- ALTER TABLE MODIFY PARTITION ADD SUBPARTITION. This statement locks the table in Row Exclusive (SX) mode and the subpartition in Exclusive (X) mode.
- ALTER TABLE COALESCE PARTITION. This statement locks the table in Row Exclusive (SX) mode and locks the last partition of the table and the partition into which rows from the last partition will be rehashed in Exclusive (X) mode.
- ALTER TABLE MODIFY PARTITION COALESCE SUBPARTITION. This statement locks the table in Row Exclusive (SX) mode and locks the last subpartition of the partition and the subpartition into which rows from the last subpartition will be rehashed in Exclusive (X) mode.
- LOAD PARTITION or SUBPARTITION. When run sequentially, this statement locks the table in Row Exclusive (SX) mode and the partition or subpartition being loaded in Exclusive (X) mode. When run in parallel, it locks the table in Row Share (SS) mode and the partition or subpartition being loaded in Shared (S) mode.
- EXPORT PARTITION or SUBPARTITION. This statement acquires no DML locks (it runs a SELECT from a partition or subpartition).
- IMPORT PARTITION or SUBPARTITION. This statement locks the table and the partition or subpartition into which data is being imported (by INSERT into a subpartition) in Row Exclusive (SX) mode.

Finally, some operations may follow either one-step or three-step protocol:

- ALTER TABLE DROP PARTITION and ALTER TABLE TRUNCATE PARTITION

  If the table being altered has no globalthen statements in this group execute using the one-step protocol and they are fast. Otherwise, they execute using the three-step protocol. In the latter case, the base table is locked in Row Exclusive (SX) mode and the partition is locked in Exclusive (X) mode.

- ALTER INDEX SPLIT PARTITION (allowed for global indexes only)

  If the partition to be split is USABLE, then the statement follows the 3-step protocol, and partitions resulting from the SPLIT are USABLE. If, on the other hand, the partition being split is UNUSABLE, then the operation follows the 1-step protocol, and resulting partitions are also marked UNUSABLE.

- ALTER INDEX DROP PARTITION (allowed for global indexes only)

  If the partition to be dropped is USABLE, then the statement follows the three-step protocol. Otherwise it follows the one-step protocol.

Conventional Path SQL*Loader and Import use SQL INSERT so they are classified as DML operations for the purposes of the model. Export uses SQL SELECT so it is classified as a query operation.

Operations That Can Run Concurrently The rules in this section can be derived from the definitions of one-step and three-step operations.

While a one-step operation is in progress:

- You can run queries on the table
- You cannot run any other operation (DDL, utility, or DML)

Because queries (READ operations) do not take DML locks, queries are allowed on a partition which is being split or moved while the SPLIT or MOVE is being processed. However, the current segments are dropped at the end of the operation, and the space may be reused. An error is signalled if the space is reused.

While an ALTER TABLE MOVE PARTITION, ALTER TABLE SPLIT PARTITION, ALTER TABLE EXCHANGE PARTITION, or Direct Path Load Table Partition is in progress on a partition:

- You can move, split, exchange, or direct path load other partitions in the same table except when performing an exchange with a hash-partitioned table. In this case, you cannot execute any DDL or DML on the hash-partitioned table.
- You can run queries on the table.
- You can execute DML operations on the table provided that they do not write to that partition.
- You can rebuild any local index partition other than the ones that correspond to that partition.
- You cannot run any maintenance operation on the table or its indexes other than the ones listed above.

While a CREATE INDEX or ALTER INDEX REBUILD PARTITION or ALTER INDEX DROP/SPLIT PARTIITON applied to a USABLE partition for a global index is in progress:

- You can run queries on the underlying table.
- You can create other indexes on the table, rebuild partitions in existing indexes, or drop or split USABLE partitions in existing indexes.
- You cannot execute any DML operation on the table or run any maintenance operation on the table or its indexes other than the ones listed above.

While an ALTER INDEX REBUILD PARTITION for a local index is in progress on a partition which corresponds to an underlying table partition:

- You can move, split, or direct path load any partition except the underlying table partition.
- You can run queries on the table.
- You can execute DML operations on the table provided that they do not write to the underlying table partition.
- You can rebuild other partitions in the index. You can also create other indexes on the table or rebuild partitions in other indexes.
- You cannot run any maintenance operation on the table or its indexes other than the ones listed above.

Some maintenance operations on a partition of a table cause the global indexes of the table or the index partitions to become UNUSABLE. An example is ALTER TABLE MOVE PARTITION. The DBA has to run a script that includes global index rebuilds in addition to the partition maintenance operation. Consequently from a user point of view these operations serialize access to the entire table. Operations such as ALTER TABLE MOVE/SPLIT PARTITION make UNUSABLE any nonpartitioned global indexes as well as all partitions of partitioned global indexes.

Note that table partition operations which mark all partitions of global indexes also mark one partition of local index (the partition corresponding to the table partition being operated on) UNUSABLE.

Similarly some partition maintenance operations require disabling Referential Integrity Constraints before the operation, and re-enabling them afterwards. An example is a ALTER TABLE DROP PARTITION of a nonempty partition. The DBA has to run a script that includes constraint re-enabling in addition to the partition maintenance operation. Consequently, from a user point of view these operations serialize access to the entire table.

Table 11–2 shows the operations that can be performed concurrently with maintenance operations on subpartitions.

Table 11–2 Concurrent Operations on Subpartitions

Maintenance Operation	Operations That Can Be Performed Concurrently
ALTER TABLE / INDEX MODIFY DEFAULT ATTRIBUTES OF PARTITION	Queries on the table
ALTER TABLE EXCHANGE SUBPARTITION WITHOUT VALIDATION	
ALTER TABLE / INDEX MODIFY SUBPARTITION (unless ALLOCATE EXTENT is specified)	
ALTER TABLE / INDEX RENAME SUBPARTITION	
ALTER TABLE MODIFY PARTITION ADD SUBPARTITION	Queries on the table
ALTER TABLE MODIFY PARTITION COALESCE SUBPARTITION	DML as long as no attempt is made to modify contents of the subpartitions affected by the statement
	Maintenance operations on other partitions and their subpartitions
	Maintenance operations on other subpartitions of the partition
	ALTER INDEX REBUILD SUBPARTITION on subpartitions of local indexes corresponding to other subpartitions of the same partition or to subpartitions of other partitions of the table

Table 11–2 Concurrent Operations on Subpartitions (Cont.)

Maintenance Operation	Operations That Can Be Performed Concurrently
ALTER TABLE EXCHANGE SUBPARTITION WITH VALIDATION	Queries on the table
ALTER TABLE / INDEX MODIFY SUBPARTITION ALLOCATE EXTENT	DML as long as no attempt is made to modify contents of the subpartition(s) referenced by the statement
ALTER TABLE MOVE SUBPARTITION	Maintenance operations on other partitions and their subpartitions except for hash partitions being exchanged
LOAD SUBPARTITION	Maintenance operations on other subpartitions of the partition except for hash partitions being exchanged
	ALTER INDEX REBUILD SUBPARTITION on subpartitions of local indexes corresponding to other subpartitions of the same partition or to subpartitions of other partitions of the table
IMPORT SUBPARTITION	Queries on the table
	DML on the table
	Maintenance operations on other partitions and their subpartitions
	Maintenance operations on other subpartitions of the partition
	ALTER INDEX REBUILD SUBPARTITION on subpartitions of local indexes corresponding to other subpartitions of the same partition or to subpartitions of other partitions of the table
EXPORT PARTITION	Any operation on the table and indexes defined on it, their partitions, and subpartitions

Table 11–2 Concurrent Operations on Subpartitions (Cont.)

Maintenance Operation	Operations That Can Be Performed Concurrently
ALTER (local) INDEX REBUILD SUBPARTITION	Queries on the table
	DML as long as no attempt is made to modify contents of the subpartition corresponding to the index subpartition being rebuilt
	Maintenance operations on subpartitions of the index's partitions, other than the partition whose subpartition is being rebuilt
	Maintenance operations on other subpartitions of the index partition
	CREATE new indexes on the underlying table
	Maintenance operations on existing indexes on the underlying table, as well as their partitions and subpartitions (if applicable)
	Maintenance operations on partitions of the underlying table, other than the partition corresponding to the index partition whose subpartition is being rebuilt
	Maintenance operations on subpartitions of the underlying table, other than the one corresponding to the index subpartition being rebuilt

Partition Maintenance Operations on Tables with LOB Columns

Table partition maintenance operations handle partitioned tables with LOB columns as follows:

- ADD PARTITION: For every LOB column in a table, a new LOB data partition and LOB index partition are created. You can specify the physical attributes of the new LOB data partitions.
- DROP PARTITION: For every LOB column in a table, the LOB data partition and LOB index partition corresponding to the table partition being dropped are also dropped.
- EXCHANGE PARTITION: The algorithm used to determine whether a given nonpartitioned table may be exchanged with a partition of a partitioned table can also handle LOB columns.

- IMPORT/EXPORT PARTITION: You can import/export partitions of tables containing LOB column(s).
- LOAD PARTITION: In addition to loading data into the table partition, data is also loaded into LOB data partitions corresponding to that table partition, while making appropriate changes to LOB index partitions.
- MODIFY PARTITION: You can modify attributes of LOB data partitions associated with a given table partition. Although you cannot specify attributes for a LOB index partition, changes to an attribute of a LOB data partition may result in changes to the corresponding attribute of a LOB index partition associated with it.
- MOVE PARTITION: For every LOB column in a table, a LOB data partition and LOB index partition corresponding to the table partition being moved can also be moved, although they do not have to be moved, for example, if a LOB data partition resides on a read-only device. You can specify new physical attributes of LOB data partitions.
- SPLIT PARTITION: For every LOB column in a table, two new LOB data partitions and LOB index partitions are created. LOB instances are divided between the new partitions based on the values of the partitioning columns in the row of which they are a part. You can specify physical attributes for the new LOB data partitions.
- TRUNCATE PARTITION: For every LOB column in a table, the LOB data partition and LOB index partition corresponding to the table partition being truncated are also truncated.

Addition of LOB columns to partitioned tables has no effect on the concurrency model for maintenance operations.

See Also:

- "Partitioning of Tables with LOB Columns" on page 11-37
- "Concurrency Model for Maintenance Operations" on page 11-49

Queries and Partition Maintenance Operations

Queries whose execution starts before invocation of a partition maintenance operation, or before dictionary updates are done during a partition maintenance operation, correctly access via Consistent Read the data of the affected partitions as existing at query snapshot time. Such queries either successfully complete returning all relevant data as present at snapshot time, or fail to complete returning error

ORA-8103 or ORA-1410. The application should reissue the query if one of these errors is returned.

Queries that use a partitioned index, and that start with some of the index partitions marked as INDEX UNUSABLE, return an error when they actually access one of these partitions for the first time. This happens even if the partition has been made USABLE after query start.

Cursor Invalidation

Although many of the new DDL statements are partition-based, cursor invalidation is still table-based. This means that any DDL statement that modifies table T also invalidates all cursors that depend on T, even if the statement affects only one partition P of T and the cursors do not access partition P.

LOGGING and NOLOGGING Operations

All partition maintenance operations can be run in LOGGING mode. However, some operations support a NOLOGGING clause:

- Parallel CREATE TABLE ... AS SELECT
- CREATE INDEX
- SPLIT, MOVE, or REBUILD PARTITION
- Direct Path SQL*Loader
- Direct-load INSERT

LOGGING is the default, except when the database operates in NOARCHIVELOG mode in which case NOLOGGING is the default. DDL and utility statements that do not support the LOGGING / NOLOGGING clause always run in recoverable mode (LOGGING).

Note: LOGGING or NOLOGGING is not an attribute of an operation but of a physical object. Hence, you cannot specify LOGGING or NOLOGGING in an INSERT statement. Instead, if you want to alter the logging mode of a table or indexes involved in an insert operation, you need to issue ALTER TABLE / INDEX [NO]LOGGING before issuing the INSERT statement.

For more information, see "Logging Mode" on page 22-5.

Managing Indexes

You can always rename, change the physical storage attributes, or rebuild a partition of a local or global index. Changing how an index is partitioned must be handled differently depending on whether the index is local or global.

Local Indexes

Oracle guarantees that the partitioning of a local index matches the partitioning of the underlying table. It does this by automatically creating or dropping index partitions as necessary when you alter the underlying table. You cannot explicitly add, drop, or split a partition in a local index.

For each local index:

- When you add a partition to the underlying table, Oracle automatically creates a new index partition with the same partition bound as the new table partition.
- When you drop a partition in the underlying table, Oracle automatically drops the corresponding index partition.
- When you split a partition in the underlying table, Oracle automatically splits the corresponding index partition. The two new index partitions have the same partition bounds as the new table partitions.

  Note that local index partitions produced as a result of splitting a parent table partition are marked UNUSABLE if a corresponding table partition is nonempty.

When Oracle creates a new local index partition by the ADD or SPLIT operations of the corresponding table partition:

- It tries to assign it the same name as the corresponding table partition. If that fails, then it generates a name with the form SYS_P*nnn*. You can rename the partition later.
- For ADD PARTITION, Oracle creates a segment with the default physical storage attributes of the base index. DEFAULT cases local index partitions to be co-located with corresponding base table partitions. If a tablespace other than DEFAULT has been specified for the parent index, then the index partition is placed in that tablespace. Otherwise, the tablespace in which the new index partition resides is that of the corresponding partition of the underlying table. You can modify these attributes later. The only way to modify TABLESPACE is to rebuild.

- For SPLIT PARTITION, attributes of the index partition being split are used for the index partitions resulting from the split. Partition names are the exception, although Oracle reuses table partition names when possible. For example, for a table partition with name TP and a local index with name IP, if TP is split into TP and TP1, then the names of the local index partitions are IP and TP1 (or a system generated name if TP1 is already in use for that index). If TP is split into TP1 and TP2, then the local index partitions are TP1 and TP2. That is, if the table partition name is reused, then Oracle tries to reuse the local index partition name also. All other attributes, however, are inherited from the index partition being split.

Rather than dropping a local index partition explicitly (for example, before loading data into its corresponding table partition), you can:

1. EXCHANGE the table partition into a nonpartitioned table, a hash-partitioned table, or a composite-partitioned table.
2. Drop the index on that table and perform your load operation.
3. Create the index and EXCHANGE the table back into the partition using the INCLUDING INDEXES clause.

> **See Also:** "Partition and Subpartition Names" on page 11-18 for more information about naming a new local index partition

Global Partitioned Indexes

The DBA is responsible for maintaining the partitioning of a global index. You can drop or split a partition in a global index. However, you cannot add a partition to a global index because the high partition of a global index always has a partition bound of MAXVALUE.

> **See Also:** "Management of Global Partitioned Indexes" on page 11-32 for more information on managing global indexes

Rebuild Index Partition

The ALTER INDEX REBUILD PARTITION statement can be used to regenerate a single partition in a local or global partitioned index. This saves you from having to perform DROP INDEX and then CREATE INDEX, which would affect all partitions in the index.

ALTER INDEX REBUILD PARTITION has four important applications:

- To recluster an index partition to recover space and improve performance.

- To repair an index partition in case of a media failure on the volume where the index partition resides or a software corruption of the segment containing the index partition.
- To regenerate a local index partition after loading the underlying table partition with Import or SQL*Loader. These utilities offer a performance option to bypass index maintenance, mark the affected index partitions INDEX UNUSABLE, and let the DBA rebuild them later. INDEX UNUSABLE is explained in the next section. In other words, the strategy of "drop index then re-create index" can be replaced by a strategy of "mark index partition UNUSABLE then rebuild index partition."
- To rebuild index partitions rendered UNUSABLE by partition maintenance operations on the underlying table.

INDEX UNUSABLE Attribute

Some maintenance operations mark indexes *INDEX UNUSABLE (IU)*. INDEX UNUSABLE is an attribute of a nonpartitioned index and of a partition in a partitioned index. When an index or index partition is marked IU, you get an error if you try to execute a SELECT or DML statement that requires the index or partition.

When a single index partition is marked IU, you must rebuild the partition to make it valid again before using it. However, while one partition is marked IU the other partitions of the index are valid and you can execute SELECT or DML statements that require the index as long as the statements do not access the IU partition.

You can also split or rename the IU partition before rebuilding it, and you can drop an IU partition of a GLOBAL index.

When a nonpartitioned index is marked IU, you can drop the index. You can also drop an IU partition of a GLOBAL index.x and re-create it, and you can use ALTER INDEX REBUILD to rebuild a nonpartitioned index.

Six types of maintenance operations can mark index partitions INDEX UNUSABLE. In all cases, you must rebuild the index partitions when the operation is complete.

- Operations like Import Partition or conventional path SQL*Loader that offer an option to bypass local index maintenance. When the Import is complete, the affected local index partitions are marked IU.
- Direct path SQL*Loader leaves affected local index partitions and global indexes in an IU state if the index is out of date with respect to the data that it

indexes. (INDEX UNUSABLE was previously known as Direct Load State.) The index can be out of date for the following reasons:

  - The index could not be maintained by the load due to a space management error (for example, out of extents).
  - The user requested the SKIP_INDEX_MAINTENANCE clause.

- Partition maintenance operations like ALTER TABLE MOVE PARTITION that change rowids. These operations mark the affected local index partition and all global index partitions IU.
- Partition maintenance operations like ALTER TABLE TRUNCATE PARTITION or DROP PARTITION that remove rows from the table. These operations mark the affected local index partition and all global index partitions IU.
- Partition maintenance operations like ALTER TABLE SPLIT PARTITION that modify the partition definition of local indexes but do not automatically rebuild the index data to match the new definitions. These operations mark the affected local index partitions IU. ALTER TABLE SPLIT PARTITION also marks all global index partitions IU because it results in changes to rowids.
- Index maintenance operations like ALTER INDEX SPLIT PARTITION that modify the partitioning definition of the index but do not automatically rebuild the affected partitions. These operations mark the affected index partitions IU. However, if you split a USABLE partition of a global index, resulting partitions are created USABLE. If the partition which was split was marked IU, then so are the partitions resulting from the split. Note that dropping a partition of a global index which is either IU or is not empty causes the next partition of the index to become IU.

Privileges for Partitioned Tables and Indexes

Privileges for partitions are granted on the parent table or index, not on individual partitions. If you want to grant access to a table on a per-partition basis, you can define a view on a partition of a table and then grant privileges on that view.

If a user or role has the privileges required to perform an Oracle operation on nonpartitioned tables and indexes, including the necessary resource privileges, then the same Oracle operations are allowed on partitioned tables and indexes. For example:

- If you can create nonpartitioned tables, then you can create partitioned tables.
- If you can drop nonpartitioned indexes, then you can drop partitioned indexes.

- If you can add a column via ALTER to nonpartitioned tables, then you can add a column via ALTER to partitioned tables.

If a user or role has the privileges required to perform an ALTER operation on a table or index, then the ALTER operations on partitions of the table or index can be invoked, with some exceptions.

See Also:

- "Viewing Partitions or Subpartitions as Tables" on page 11-64
- *Oracle8i SQL Reference* for information about privileges for the ALTER TABLE and ALTER INDEX statements

Auditing for Partitioned Tables and Indexes

All of the ALTER TABLE PARTITION operations are audited just like ALTER TABLE operations. No additional audit attributes are used for partitions.

Partition-Extended and Subpartition-Extended Table Names

You can perform bulk operations at the partition or subpartition level. That is, bulk operations can be restricted to just the rows of a particular partition or subpartition. For example, if you want to drop a partition without making all the global indexes UNUSABLE, you would want to delete all the rows from just that partition.

Such operations are very naturally expressed with a SQL extension that provides syntax for partition-extended table names and subpartition-extended table names. Trying to phrase the same operations with a WHERE clause predicate is often cumbersome, especially when a range partitioning key uses multiple columns.

PARTITION and SUBPARTITION Specifications

The table specification syntax for the following DML statements can contain an optional PARTITION specification for partitioned tables, or an optional PARTITION or SUBPARTITION specification for composite-partitioned tables:

- SELECT
- INSERT
- UPDATE
- DELETE
- LOCK TABLE

For example:

```
SELECT * FROM schema.table PARTITION part_name;
```

For a composite-partitioned table, using the PARTITION specification restricts the operation to data contained in all subpartitions of the specified partition.

> **See Also:** *Oracle8i SQL Reference* for information about the syntax of DML statements

Viewing Partitions or Subpartitions as Tables

The PARTITION or SUBPARTITION syntax for table specifications provides a simple way of viewing individual partitions or subpartitions as tables: You can use the partition-extended table name or subpartition-extended table name to create a view that selects from just one partition or subpartition, and this view can then be used in place of a table. For example:

```
CREATE VIEW sales_feb98_v1 AS
  SELECT * FROM sales SUBPARTITION (feb98_s1);

SELECT * FROM sales_feb98_v1;
```

With such views you can also build partition-level or subpartition-level access control mechanisms by granting or revoking privileges on these views to or from other users or roles.

> **Note:** For application portability and ANSI syntax compliance, you should always use views to insulate your applications from this Oracle proprietary extension.

Using Partition- and Subpartition-Extended Table Names

This section describes restrictions on the use of the PARTITION and SUBPARTITION clauses in table specifications, and provides examples of SQL statements that include the PARTITION or SUBPARTITION clause.

Restrictions on Partition- and Subpartition-Extended Table Names

The use of partition- and subpartition-extended table names has the following restrictions:

- *A partition- or subpartition-extended table name cannot refer to a remote schema object.*

A partition- or subpartition-extended table name cannot contain a database link or a synonym which translates to a table with a database link. If you need to use remote partitions or subpartitions, you can create a view at the remote site which uses the partition- or subpartition-extended table name syntax and refer to that remote view.

- *The partition- or subpartition-extended table name syntax is not supported by PL/SQL.*

  A SQL statement that has the partition- or subpartition-extended table name syntax cannot be used in a PL/SQL block, although it can be used through dynamic SQL with the DBMS_SQL package. If you need to refer to a partition or subpartition within a PL/SQL block, you can use views which in turn use the partition- or subpartition-extended table name syntax.

- *Only base tables are allowed.*

  A partition or subpartition extension must be specified with a base table. No synonyms, views, or any other schema objects are allowed.

Examples of Using the PARTITION Specification

The following statements contain valid partition-extended table names:

```
SELECT * FROM sales PARTITION (nov97) s
  WHERE s.amount_of_sale > 1000;

UPDATE sales PARTITION (feb98) s
  SET s.account_name = UPPER(s.account_name);

DELETE FROM sales PARTITION (nov97)
  WHERE amount_of_sale != 0;

INSERT INTO sales PARTITION (oct97)
  SELECT * FROM lastest_data;

INSERT INTO sales PARTITION (oct97)
  VALUES (...);

INSERT INTO sales PARTITION (oct97)
  (acct_no, ..., week_no)
  VALUES (...);

LOCK TABLE sales PARTITION (jun98) IN EXCLUSIVE MODE;

CREATE VIEW sales_feb98 AS
  SELECT * FROM sales PARTITION (feb98);
```

Examples of Using the SUBPARTITION Specification

The following statements contain valid subpartition-extended table names:

```
SELECT * FROM sales SUBPARTITION (nov97_s1) s
  WHERE s.amount_of_sale > 1000;

UPDATE sales SUBPARTITION (feb98_s4) s
  SET s.account_name = UPPER(s.account_name);

DELETE FROM sales SUBPARTITION (nov97_s3)
  WHERE amount_of_sale != 0;

INSERT INTO sales SUBPARTITION (oct97_s5)
  SELECT * FROM lastest_data;

INSERT INTO sales SUBPARTITION (oct97_s2)
  VALUES (...);

INSERT INTO sales SUBPARTITION (oct97_s4)
  (acct_no, ..., week_no)
  VALUES (...);

LOCK TABLE sales SUBPARTITION (jun98_s1) IN EXCLUSIVE MODE;

CREATE VIEW sales_feb98_1 AS
  SELECT * FROM sales SUBPARTITION (feb98_s1);
```

12

Built-In 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
- 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 the *PL/SQL User's Guide and Reference* for information about PL/SQL datatypes.

Table 12–1 summarizes the characteristics of each Oracle datatype.

Table 12–1 Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length and Default
CHAR (*size*)	Fixed-length character data of length *size* bytes.	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (one-byte or multibyte) before setting *size*.
VARCHAR2 (*size*)	Variable-length character data. A maximum *size* must be specified.	Variable for each row, up to 4000 bytes per row. Consider the character set (one-byte or multibyte) before setting *size*.
NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the national character set.	Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (*size*)	Variable-length character data of length *size* characters or bytes, depending on national character set. A maximum *size* must be specified.	Variable for each row. Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
LONG	Variable-length character data.	Variable for each row in the table, up to 2^{31} - 1 bytes, or 2 gigabytes, per row.
NUMBER (*p*, *s*)	Variable-length numeric data. Maximum precision *p* and / or scale *s* is 38.	Variable for each row. The maximum space required for a given column is 21 bytes per row.
DATE	Fixed-length date and time data, ranging from January 1, 4712 BCE to December 31, 4712 CE ("A.D.")	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.

Table 12–1 Summary of Oracle Built-In Datatypes (Cont.)

Datatype	Description	Column Length and Default
RAW (*size*)	Variable-length raw binary data. A maximum *size* must be specified.	Variable for each row in the table, up to 2000 bytes per row.
LONG RAW	Variable-length raw binary data.	Variable for each row in the table, up to 2^{31} - 1 bytes, or 2 gigabytes, per row.
BLOB	Binary data.	Up to 2^{32} - 1 bytes, or 4 gigabytes.
CLOB	Single-byte character data.	Up to 2^{32} - 1 bytes, or 4 gigabytes.
NCLOB	Single-byte or fixed- or variable-width multibyte national character set (NCHAR) data.	Up to 2^{32} - 1 bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file.	Up to 2^{32} - 1 bytes, or 4 gigabytes.
ROWID	Binary data representing a physical row address.	Fixed at 10 bytes (extended rowid) or 6 bytes (restricted rowid) for each row in the table.
UROWID	Binary data representing any type of row address: physical, logical, or foreign.	Up to 4000 bytes (but for a logical rowid, only 3950 bytes can be used for the primary key). The default is 4000 bytes.

The sections that follow describe each of the built-in datatypes in more detail.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for information about how to use the built-in datatypes

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 and never changes. Examples of character sets are 7-bit ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), Code Page 500, and Japan Extended UNIX. Oracle supports both single-byte and multibyte encoding schemes.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for information about how to select a character datatype
- *Oracle8i National Language Support Guide* for more information about converting character data. If you are converting data from one character set to another, you must ensure that [CHAR | VARCHAR] data consists of well-formed strings.

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, not characters) between 1 and 2000 for the CHAR column width. The default is 1. 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: *Oracle8i 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, not characters) between 1 and 4000 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.

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:** *Oracle8i SQL Reference* for details about nonpadded comparison semantics

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. However, in a future version of Oracle, VARCHAR might store variable-length character strings compared with different comparison semantics. Therefore, to avoid possible changes in behavior you should always use the VARCHAR2 datatype to store variable-length character strings.

Column Lengths for Character Datatypes and NLS Character Sets

The Oracle National Language Support (NLS) feature allows the use of various character sets for the character datatypes. National Language Support enables you to process single-byte and multi-byte character data and convert between character sets. Client sessions can use national character sets different from the database character set.

You should 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.

> **See Also:** *Oracle8i National Language Support Guide* for more information about the NLS feature of Oracle

NCHAR and NVARCHAR2 Datatypes

The NCHAR and NVARCHAR2 datatypes store NLS character data. The NCHAR datatype stores fixed-length character strings that correspond to a fixed-length or variable-length national character set. The NVARCHAR2 datatype stores variable-length character strings.

When you create a table with an NCHAR or NVARCHAR2 column, you specify a maximum size that is either the number of characters (for a fixed-length national character set) or the number of bytes (for a variable-length national character set).

- The maximum length for an NCHAR column is 2000 bytes, or the number of characters that can be stored in 2000 bytes.
- The maximum length for an NVARCHAR2 column is 4000 bytes, or the number of characters that can be stored in 4000 bytes.

> **See Also:** *Oracle8i National Language Support Guide* for more information about the NCHAR and NVARCHAR2 datatypes

LOB Character Datatypes

The LOB datatypes for character data are CLOB and NCLOB. They can store up to four gigabytes of character data (CLOB) or national character set data (NCLOB).

> **See Also :** "LOB Datatypes" on page 12-12

LONG Datatype

Columns defined as LONG can store variable-length character data containing up to two 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.

Note: The LONG datatype is provided for backward compatibility with existing applications. In new applications, you should use CLOB and NCLOB datatypes for large amounts of character data.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for information about the restrictions on the LONG datatype
- "RAW and LONG RAW Datatypes" on page 12-15 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 12–2 shows examples of how data would be stored using different scale factors.

Table 12–2 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 should round to the nearest hundredths, as shown in Table 12–2.

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 one 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 4.12×10^2, with one byte used to store the exponent (2) and two 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 one byte; positive infinity requires two 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 below:

```
'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 above.

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.'));
```

Using 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 may 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 just 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 properly with four-digit years.

However, some applications might be written with an assumption about the year (such as assuming that everything is 19xx). Application programmers should therefore review and test their code with regard to the year 2000.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for more information about centuries and date format masks
- *Oracle8i SQL Reference* for information about date format codes

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 four gigabytes in size. They provide efficient, random, piece-wise access to the data.

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 four gigabytes, but the maximum size of a LONG is two 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.

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).

See Also:

- "Default Logging Mode" on page 22-8 for information about the LOB attributes LOGGING and NOLOGGING
- *Oracle8i SQL Reference* for a complete list of differences between the LOB datatypes and the LONG and LONG RAW datatypes
- *Oracle8i 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 four 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 four gigabytes of character data in the database. CLOBs store single-byte character set data and NCLOBs store fixed-width and varying-width multibyte national character set data (NCHAR data).

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.

The CLOB or NCLOB 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 or NCLOB column using a varying-width character set, Oracle converts the data into Unicode before storing it in the database.

You cannot create an object type with NCLOB attributes, but you can specify NCLOB parameters in a method for an object type.

> **See Also:** *Oracle8i National Language 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 four 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 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, you should 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 that Net8 (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, Net8 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:** *Oracle8i 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 12-23

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. 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..

See Also: "How Rowids Are Used" on page 12-20

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 essentially 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* 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;
```

might 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:** *Oracle8i 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;
```

might return the following row information:

```
ROWID              ENAME
------------------ ----------
00000DD5.0000.0001 KRISHNAN
00000DD5.0001.0001 ARBUCKLE
00000DD5.0002.0001 NGUYEN
```

As shown above, 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 Using Rowids

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:

- *Oracle8i SQL Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle8i Designing and Tuning for Performance*

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 may 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 might 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 12-15

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.

You should collect index statistics with the DBMS_STATS package or ANALYZE statement to keep track of the staleness of guesses, so that 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), you should 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:** "Statistics for Cost-Based Optimization" on page 21-9 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 Open 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:

- The relevant manual for OCIs or precompilers for further details on handling rowids with non-Oracle systems
- "ROWID and UROWID Datatypes" on page 12-15

ANSI, DB2, and SQL/DS Datatypes

The ANSI datatype conversions to Oracle datatypes are shown in Table 12–3. The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, *s* (scale) defaults to 0.

Table 12–3 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 12–4 shows the DB2 and SQL/DS conversions.

Table 12–4 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

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 using one of the following functions:

- TO_NUMBER()
- TO_CHAR()
- TO_DATE()
- CHARTOROWID()
- ROWIDTOCHAR()
- HEXTORAW()
- RAWTOHEX()
- REFTOHEX()

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for the rules for implicit datatype conversions

13

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 Datatypes
- Application Interfaces

Introduction to User-Defined Datatypes

Relational database management systems (RDBMSs) are the standard tool for managing business data. They provide fast, efficient, and completely reliable access to huge amounts of data for millions of businesses around the world every day.

Oracle8*i* 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 like 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.

An Example

For example, your firm may 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 like a purchase order. Some attributes, such as the list of line items, may 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.

Since the logic of the purchase order's structure and behavior is in your schema, your applications don't need to know the details and don't 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, raphics, and spatial data are examples of important business entities that do not fit neatly into those basic types. Oracle8*i* Enterprise Edition supports modeling and implementation of these complex datatypes.

User-Defined Datatypes

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 12, "Built-In Datatypes"
- *Oracle8i 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 and stored in the database, or written in a language like 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 might 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 doesn't 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 might 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" on page 13-12
- *Oracle8i Application Developer's Guide - Object-Relational Features* for a complete purchase order example
- "Row Objects and Column Objects" on page 13-8

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. Such methods may be invoked in a "selfish" style, as in OBJECT.METHOD(). Member methods are useful for writing observer or mutator methods.
- A *Static* method is a function or a procedure that does not have an implicit SELF parameter. Such methods may 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:** *Oracle8i 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 may 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:** *Oracle8i 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 multi-column 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 multi-column 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 sixteen bytes as the OID for each row object by default. Oracle provides no documentation of or access to the internal structure of object identifiers. This structure can change at any time.

The OID column of an object table is a hidden column. While 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 which, as discussed below, 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 sixteen 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

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 14, "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 REFs.

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.

Dereferencing 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 above expression is a shortened form of:

```
y.name, where y = DEREF(x.manager)
```

Obtaining 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:** *Oracle8i 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 ten 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: *Oracle8i Application Developer's Guide - Object-Relational Features* for more information about using VARRAYs

Nested Tables

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 multi-column table, with a column for each attribute of the object type.

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:

- *Oracle8i Reference* for information about nested cursors
- *Oracle8i Application Developer's Guide - Object-Relational Features* for more information about using nested tables

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 REFs

> **See Also:** *Oracle8i SQL Reference* for a complete description of SQL syntax

PL/SQL

PL/SQL is a procedural language that extends SQL. It offers features like 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 13-14
> - *Pro*C/C++ Precompiler Programmer's Guide* for a complete description of the Pro*C precompiler

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 REFs
- Convert Oracle dates, strings and numbers to C data types
- Manage the size of the object cache's memory

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: *Oracle8i 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: *Oracle8i 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.

See Also: *Oracle8i Java Developer's Guide*

14

Object Views

This chapter describes object views. It contains the following major sections:

- Introduction to Object Views
- Defining Object Views
- Using Object Views
- Updating Object Views

> **See Also:** *Oracle8i 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 might use an object view to provide a version of an employee object table that does not have attributes containing sensitive data and doesn't 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.

Object views provide for co-existence of relational and object-oriented applications. 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.

Defining Object Views

Conceptually, the procedure for defining an object view is simple:

1. Define an object type to be represented by rows of the object view.
2. Write a query that specifies which data in which relational tables contain the attributes for objects of that type.
3. Specify 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 step:

4. Write an INSTEAD OF trigger procedure for Oracle to execute whenever an application program tries to update data in the object view.

After these steps 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  (
    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: "Updating Object Views" on page 14-5 for more information about writing an INSTEAD OF trigger

Using Object Views

Data in the rows of an object view may 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

Updating 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:

- *Oracle8i Application Developer's Guide - Object-Relational Features* for a purchase order/line item example that uses an INSTEAD OF trigger
- Chapter 19, "Triggers"

Updating 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. Note that 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: *Oracle8i Application Developer's Guide - Fundamentals* for a purchase order/line item example that uses an INSTEAD OF trigger on a nested table column

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:

15

SQL and PL/SQL

This chapter provides an overview of the Structured Query Language (SQL) and PL/SQL, Oracle's procedural extension to SQL. The chapter includes:

- Introduction to Structured Query Language
- SQL Processing
- PL/SQL
- PL/SQL Server Pages

See Also:

- *Oracle8i SQL Reference*
- *PL/SQL User's Guide and Reference*

Introduction to Structured Query Language

SQL is a simple, yet powerful, database access language. SQL is a 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. Oracle8*i* is broadly compatible with the SQL-99 Core specification. However, some SQL-99 Core features are not currently implemented in Oracle8*i* or differ from the Oracle8*i* implementation. Oracle Corporation is committed to fully supporting SQL-99 Core functionality in a future release, while providing upward compatibility for existing applications.

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, Oracle Enterprise Manager, and Server 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.

Although some Oracle tools and applications simplify or mask the use of SQL, all database operations are performed using SQL. Any other data access method would circumvent the security built into Oracle and potentially compromise data security and integrity.

This section includes the following topics:

- SQL Statements
- Identifying Nonstandard SQL
- Recursive SQL
- Cursors
- Shared SQL
- Parsing

See Also:

- *Oracle8i 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

SQL Statements

All operations performed on the information in an Oracle database are executed using SQL *statements*. A SQL statement is a specific instance of a valid *SQL statement*. 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 can be thought of as a simple but powerful 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 statements (DML)
- Data definition language statements (DDL)
- Transaction control statements
- Session control statements
- System control statements
- Embedded SQL statements

See Also:

- Chapter 17, "Procedures and Packages"
- Chapter 19, "Triggers"

for more information about using SQL statements in PL/SQL program units

Data Manipulation Language Statements

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)
- Add new rows of data into a table or view (INSERT)
- Change column values in existing rows of a table or view (UPDATE)
- 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

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)

- Gather statistics about schema objects, validate object structure, and list chained rows within objects (ANALYZE)
- 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 26, "Controlling Database Access"
- Chapter 27, "Privileges, Roles, and Security Policies"
- Chapter 28, "Auditing"

for more information about DDL statements that correspond to database access

Transaction Control Statements

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

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

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), to kill a session, and to perform other tasks.

Embedded SQL Statements

Embedded SQL statements incorporate DDL, DML, and transaction control statements within a procedural language program. They are used with the Oracle precompilers. Embedded SQL 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)

Identifying 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

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 for the parsing of 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 may execute several recursive calls. These recursive cursors also use *shared SQL areas*.

Shared SQL

Oracle automatically notices when applications send identical 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. Since 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 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:** *Oracle8i Application Developer's Guide - Fundamentals* for more information on 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 it is desirable to perform them as seldom as possible.

This discussion applies also to the parsing of PL/SQL blocks and allocation of PL/SQL areas. Stored procedures, functions, and packages and triggers are assigned

PL/SQL areas. Oracle also assigns each SQL statement within a PL/SQL block a shared and a private SQL area.

See Also: "PL/SQL" on page 15-16

SQL Processing

This section introduces the basics of SQL processing. Topics include:

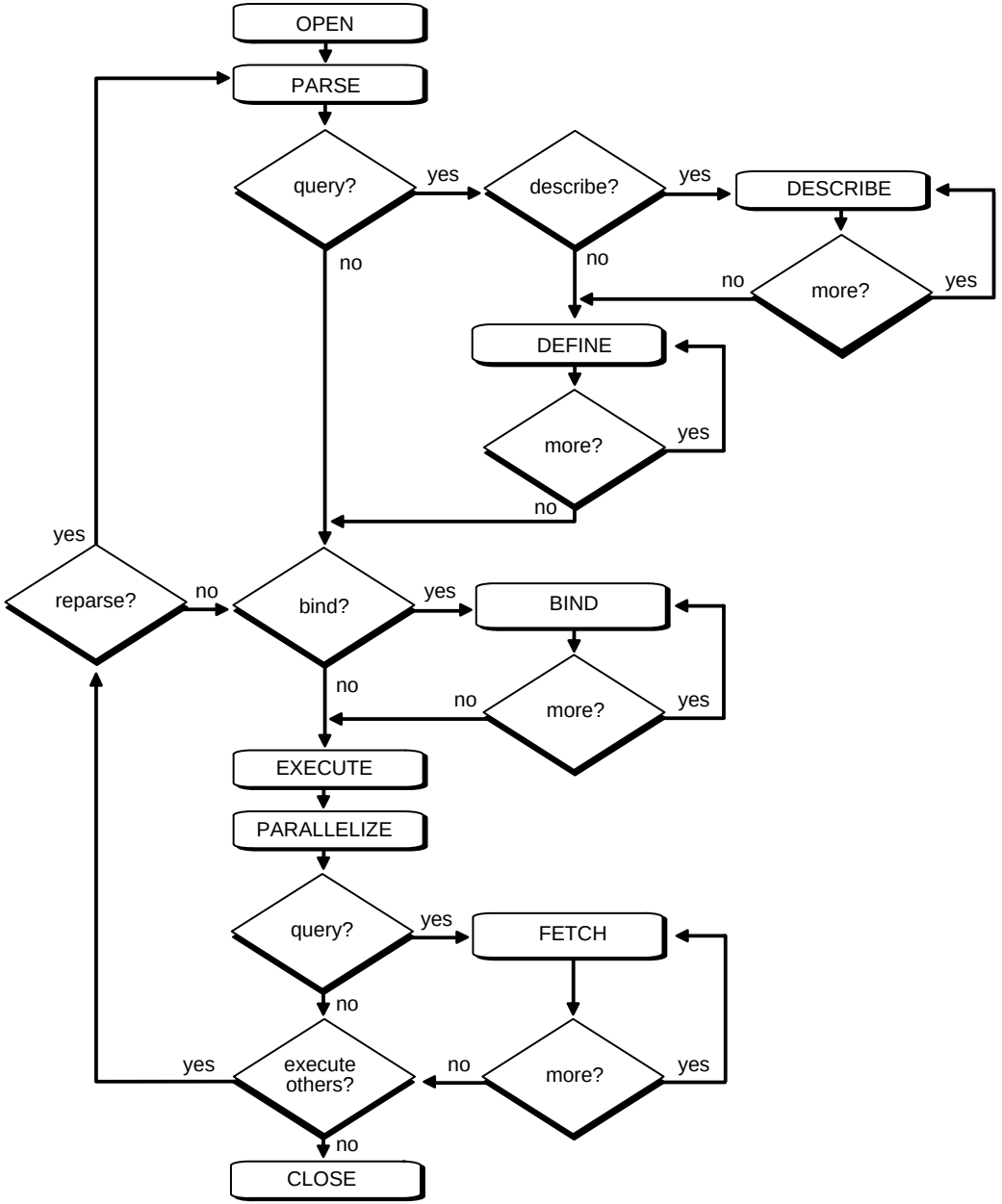
- Overview of SQL Statement Execution
- DML Statement Processing
- DDL Statement Processing
- Controlling Transactions

Overview of SQL Statement Execution

Figure 15–1 outlines the stages commonly used to process and execute a SQL statement. In some cases, Oracle might 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 might find this information useful when writing Oracle applications.

Figure 15–1 The Stages in Processing a SQL Statement



DML Statement Processing

This section provides a simple 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 might 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 (SELECTs) require several additional stages, as shown in Figure 15–1:

- Stage 3: Describe Results of a Query
- Stage 4: Define Output of a Query
- Stage 8: Fetch Rows of a Query
- Stage 9: Close the Cursor

See Also: "Query Processing" on page 15-13

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 identical 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 the parsing of 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 15-7 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 might be unaware that they are specifying bind variables, because the Oracle utility might simply prompt them for a new value.

Because you specify the location (binding by reference), you need not rebind the variable before re-execution. 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, 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 that it can complete faster.

See Also: Chapter 23, "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.

Controlling 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 properly 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 16-9

PL/SQL

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:

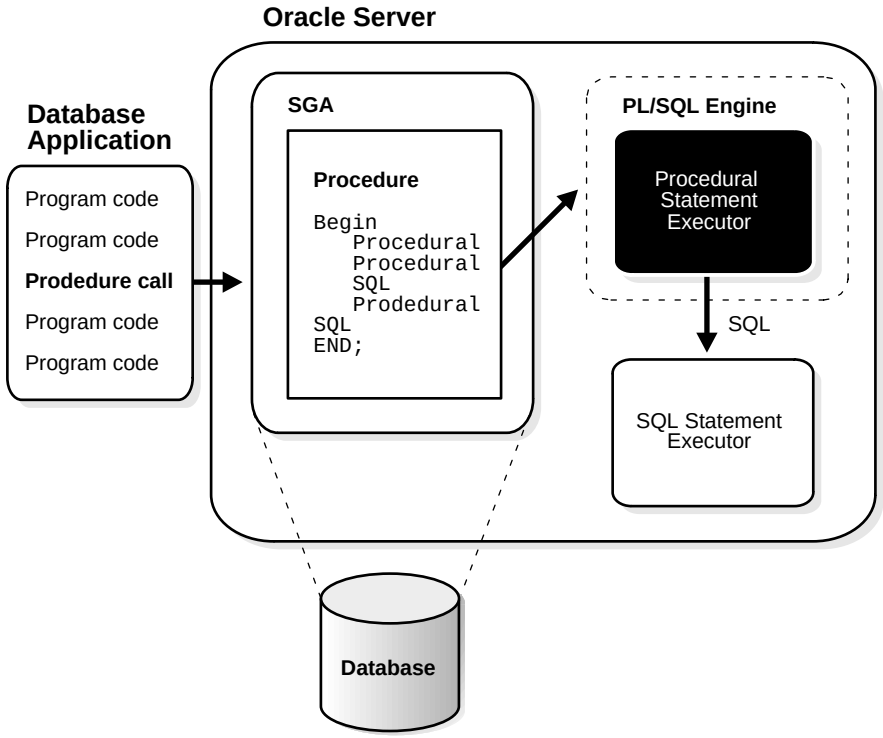
- Chapter 17, "Procedures and Packages"
- Chapter 19, "Triggers"

How PL/SQL Executes

The *PL/SQL engine*, which processes PL/SQL program units, is a special component of many Oracle products, including the Oracle server.

Figure 15–2 illustrates the PL/SQL engine contained in Oracle server.

Figure 15–2 The PL/SQL Engine and the Oracle Server



The procedure (or package) is stored in a database. When an application calls a procedure stored in the database, Oracle loads the compiled procedure (or package) into the shared pool in the system global area (SGA), and 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
- Server Manager
- 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.

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 specifically 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
- Server Manager
- 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:

- Chapter 17, "Procedures and Packages" for more information about calling a stored procedure from another PL/SQL block
- *Pro*C/C++ Precompiler Programmer's Guide* for information on how to call stored C or C++ procedures
- *Pro*COBOL Precompiler Programmer's Guide*. for information on how to call stored COBOL procedures
- Other appropriate programmer's guides for more information on how to call stored procedures of specific kinds of applications

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 data manipulation language (DML) or data definition language (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 might 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:

- *Oracle8i 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

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:** *Oracle8i Application Developer's Guide - Fundamentals* for more information about external procedures and Inter-Language Method Services

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. With the evolution of the Internet, users demand rich, up-to-date content. 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.

Scripting is easy to learn and use. 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 being embedded in the HTML pages, are executed on the server side. They fetch and manipulate data and produce HTML content which 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: *Oracle8i Application Developer's Guide - Fundamentals* for more information about PL/SQL Server Pages

16

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.

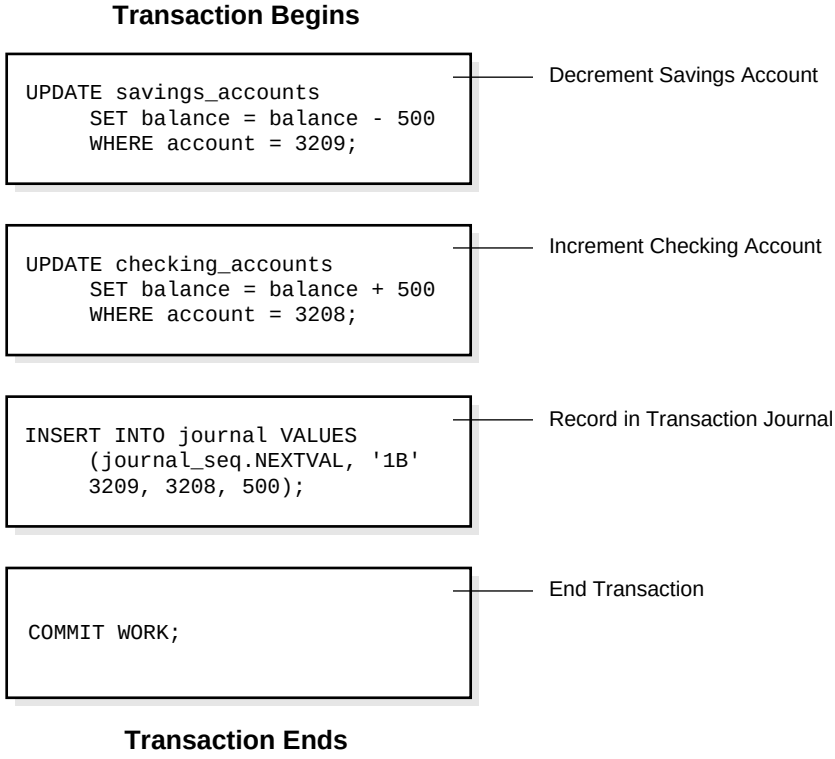
To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations:

- 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 16–1 illustrates the banking transaction example.

Figure 16–1 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 might be made via 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.

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. 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. Single SQL statements involved in a *deadlock* (competition for the same data) may also 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 24-17

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, 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 4-22

Committing 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 actually 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 may happen before the transaction commits or, alternatively, it may 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:

- "Locking Mechanisms" on page 24-3
- "Oracle Processes" on page 8-5 for more information about the background processes LGWR and DBW*n* for changes to rollback segments

Rolling Back 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" on page 16-7
- "Locking Mechanisms" on page 24-3
- *Oracle8i Recovery Manager User's Guide and Reference* for information about what happens to committed and uncommitted changes during recovery

Savepoints

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 that 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 reexecute 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.

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 up any locked resources that may be 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:

- "Distributed Transactions" on page 30-35
- *Oracle8i Distributed Database Systems*

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, since 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 may therefore be issued against tables that have the NOLOGGING attribute set.

See Also:

- "Logging Mode" on page 22-5
- *Oracle8i Designing and Tuning for Performance* for more information on discrete transactions
- *Oracle8i Supplied PL/SQL Packages 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 may 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*

17

Procedures and Packages

This chapter discusses the procedural capabilities of Oracle. It includes:

- Introduction to Stored Procedures and Packages
- Procedures and Functions
- Packages
- How Oracle Stores Procedures and Packages
- How Oracle Executes Procedures and Packages

> **See Also:** Chapter 20, "Oracle Dependency Management" 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" on page 15-16

Stored Procedures and Functions

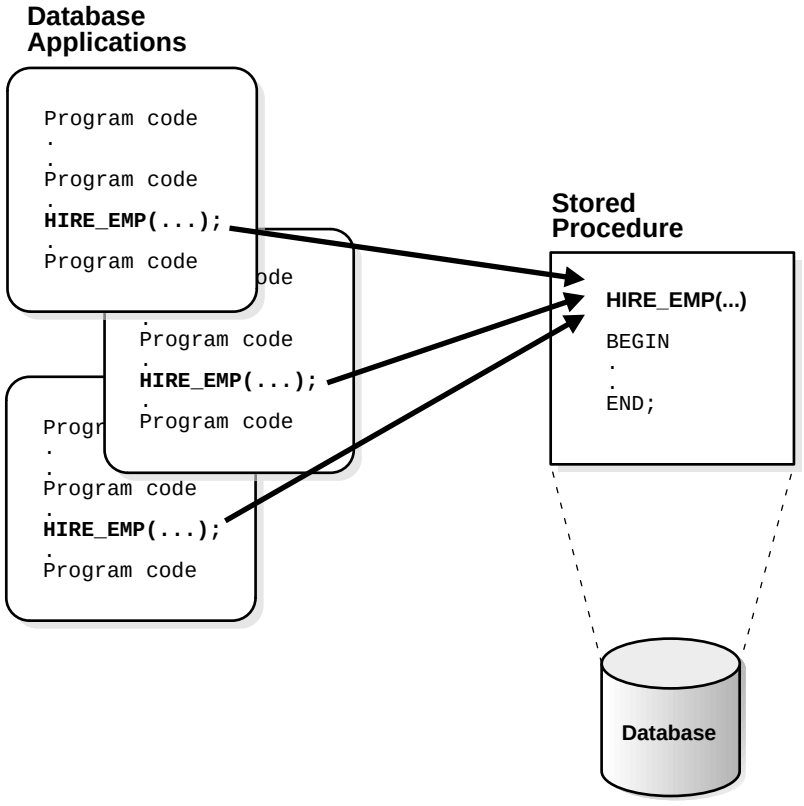
Procedures and functions are schema objects that logically group a set of SQL and other PL/SQL programming language statements together to perform a specific task. Procedures and functions are created in a user's schema and stored in a database for continued use. You can execute a procedure or function 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 17–1 illustrates a simple procedure that is stored in the database and called by several different database applications.

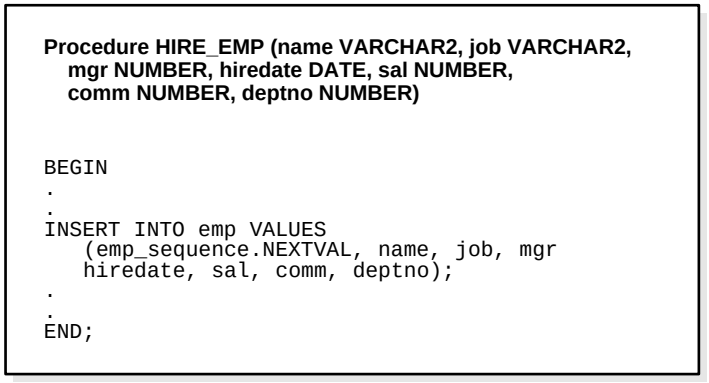
Procedures and functions are identical except that functions always return a single value to the caller, while procedures do not. For simplicity, *procedure* as used in the remainder of this chapter means *procedure or function*.

Figure 17–1 A Stored Procedure



The stored procedure in Figure 17–1, which inserts an employee record into the EMP table, is shown in Figure 17–2.

Figure 17–2 The HIRE_EMP Procedure



All of the database applications in Figure 17–1 call the HIRE_EMP procedure. Alternatively, a privileged user might 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.

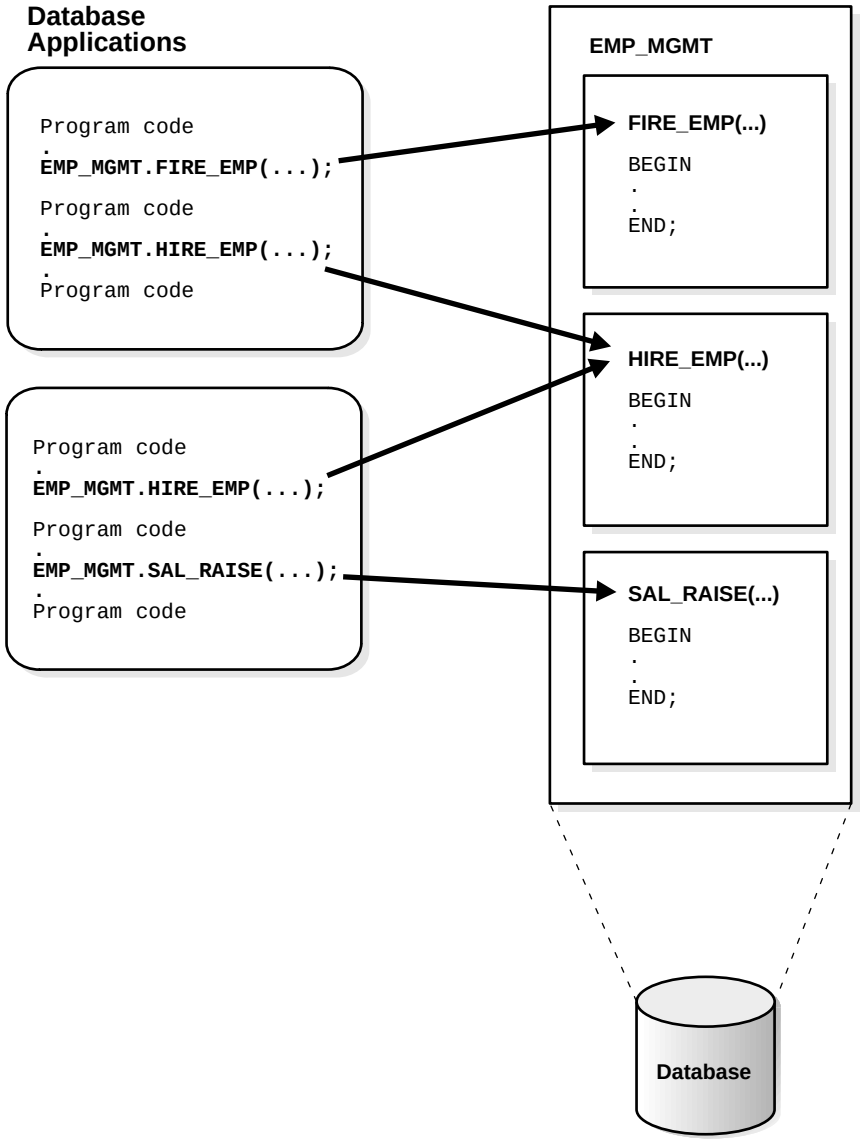
See Also: *PL/SQL User's Guide and Reference*

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 17–3 illustrates a package that encapsulates a number of procedures used to manage an employee database.

Figure 17–3 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 might 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: "Packages" on page 17-12

Procedures and Functions

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. For example, the following statement creates the CREDIT_ACCOUNT procedure, which credits money to a bank account:

```
CREATE PROCEDURE credit_account
      (acct NUMBER, credit NUMBER) AS
/* This procedure accepts two arguments: an account number and an
   amount of money to credit to the specified account. If the
   specified account does not exist, a new account is created. */

        old_balance  NUMBER;
        new_balance  NUMBER;
   BEGIN
        SELECT balance INTO old_balance FROM accounts
          WHERE acct_id = acct
          FOR UPDATE OF balance;

        new_balance := old_balance + credit;
        UPDATE accounts SET balance = new_balance
          WHERE acct_id = acct;
        COMMIT;
```

```
        EXCEPTION
          WHEN NO_DATA_FOUND THEN
            INSERT INTO accounts (acct_id, balance)
                VALUES(acct, credit);
          WHEN OTHERS THEN
        ROLLBACK;
END credit_account;
```

Notice that this sample procedure includes both SQL and PL/SQL statements.

See Also: *PL/SQL User's Guide and Reference*

Definer Rights and Invoker Rights

A PL/SQL procedure can be executed with the privileges of its owner *(definer rights)* or with the privileges of the current user *(invoker rights)*, depending on the procedure definition.

- A definer-rights procedure executes with its owner's privileges. Roles are disabled in a definer-rights procedure and in any procedure invoked directly or indirectly by a definer-rights procedure.
- An invoker-rights procedure executes with all of the invoker's privileges, including currently enabled roles. If the invoker-rights procedure was invoked directly or indirectly by a definer-rights procedure, no roles are enabled.

See Also:

- "Procedure Security Topics" on page 27-7 for more information about privileges
- "PL/SQL Blocks and Roles" on page 27-20 for more information about roles

The Current User

The *current user* starts out as the session user, who is either the logged-in user or the user associated with the remote procedure call session. On entering or exiting an invoker-rights procedure or an anonymous block, the current user does not change.

On entering a definer-rights procedure, the owner of that procedure becomes the current user. On exiting a definer-rights procedure, the current user reverts to the previous current user, that is, the current user before entry to the definer-rights procedure.

Resolution of External References

An *external reference* in a PL/SQL procedure is a name that refers to an object outside the program unit:

- For a definer-rights procedure, all external references are resolved in the schema that contains the procedure.
- For an invoker-rights procedure, external references are resolved differently depending on the kind of statement they appear in. The following external references are resolved at runtime in the schema associated with the current user. At runtime, data access through these external references is also checked against the current user's privileges:
  - External references in DML statements, including references to tables, views, sequences, and functions
  - External references in dynamic SQL statements and DBMS_SQL statements

  All external references in other PL/SQL statements including direct PL/SQL procedure calls are resolved in the schema containing the invoker-rights procedure.

Name resolution in the invoker's schema allows applications to access user-specific tables by not specifying the schema.

See Also: "Name Resolution for Database Objects and Program Units" on page 17-20

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 17-11

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 to 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 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 might 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 versus 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 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.

> **See Also:** "How Oracle Stores Procedures and Packages" on page 17-17

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" on page 17-12 for information on 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 20, "Oracle Dependency Management" 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:** *Oracle8i Application Developer's Guide - Fundamentals* for more information about external procedures and Inter-Language Method Services

Packages

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.

The following example creates the specification and body for a package that contains several procedures and functions that process banking transactions.

```
CREATE PACKAGE bank_transactions (null) AS
  minimum_balance  CONSTANT NUMBER := 100.00;
  PROCEDURE apply_transactions;
  PROCEDURE enter_transaction (acct   NUMBER,
                               kind   CHAR,
                               amount NUMBER);
END bank_transactions;

CREATE PACKAGE BODY bank_transactions AS

/*  Package to input bank transactions */
```

```
   new_status  CHAR(20);  /* Global variable to record status
                            of transaction being applied. Used
                            for update in APPLY_TRANSACTIONS. */

   PROCEDURE do_journal_entry (acct NUMBER,
                               kind CHAR) IS

/*  Records a journal entry for each bank transaction applied
    by the APPLY_TRANSACTIONS procedure. */

  BEGIN
        INSERT INTO journal
          VALUES (acct, kind, sysdate);
        IF kind = 'D' THEN
          new_status := 'Debit applied';
        ELSIF kind = 'C' THEN
          new_status := 'Credit applied';
        ELSE
          new_status := 'New account';
        END IF;
  END do_journal_entry;

  PROCEDURE credit_account (acct NUMBER, credit NUMBER) IS

/* Credits a bank account the specified amount. If the account
   does not exist, the procedure creates a new account first. */

        old_balance  NUMBER;
        new_balance  NUMBER;

  BEGIN
        SELECT balance INTO old_balance FROM accounts
          WHERE acct_id = acct
          FOR UPDATE OF balance; /* Locks account for credit update */

        new_balance := old_balance + credit;
        UPDATE accounts SET balance = new_balance
          WHERE acct_id = acct;
        do_journal_entry(acct, 'C');

  EXCEPTION
        WHEN NO_DATA_FOUND THEN  /* Create new account if not found */
          INSERT INTO accounts (acct_id, balance)
            VALUES(acct, credit);
```

```
        do_journal_entry(acct, 'N');
      WHEN OTHERS THEN  /* Return other errors to application */
        new_status := 'Error: ' || SQLERRM(SQLCODE);
  END credit_account;

  PROCEDURE debit_account (acct  NUMBER, debit NUMBER) IS

/* Debits an existing account if result is greater than the
   allowed minimum balance. */

      old_balance         NUMBER;
      new_balance         NUMBER;
      insufficient_funds  EXCEPTION;

  BEGIN
      SELECT balance INTO old_balance FROM accounts
        WHERE acct_id = acct
        FOR UPDATE OF balance;
      new_balance := old_balance - debit;
      IF new_balance >= minimum_balance THEN
        UPDATE accounts SET balance = new_balance

          WHERE acct_id = acct;
      do_journal_entry(acct, 'D');
      ELSE
        RAISE insufficient_funds;
      END IF;

  EXCEPTION
      WHEN NO_DATA_FOUND THEN
        new_status := 'Nonexistent account';
      WHEN insufficient_funds THEN
        new_status := 'Insufficient funds';
      WHEN OTHERS THEN  /* Returns other errors to application */
        new_status := 'Error: ' || SQLERRM(SQLCODE);
  END debit_account;

  PROCEDURE apply_transactions IS

/* Applies pending transactions in the table TRANSACTIONS to the
   ACCOUNTS table. Used at regular intervals to update bank
   accounts without interfering with input of new transactions. */

/* Cursor fetches and locks all rows from the TRANSACTIONS
   table with a status of 'Pending'. Locks released after all
```

```
         pending transactions have been applied. */

        CURSOR trans_cursor IS
          SELECT acct_id, kind, amount FROM transactions
            WHERE status = 'Pending'
            ORDER BY time_tag
            FOR UPDATE OF status;

   BEGIN
        FOR trans IN trans_cursor LOOP   /* implicit open and fetch */
          IF trans.kind = 'D' THEN
            debit_account(trans.acct_id, trans.amount);
          ELSIF trans.kind = 'C' THEN
            credit_account(trans.acct_id, trans.amount);
          ELSE
            new_status := 'Rejected';
          END IF;
          /* Update TRANSACTIONS table to return result of applying
             this transaction. */
          UPDATE transactions SET status = new_status
            WHERE CURRENT OF trans_cursor;
        END LOOP;
        COMMIT;  /* Release row locks in TRANSACTIONS table. */
  END apply_transactions;
  PROCEDURE enter_transaction (acct   NUMBER,
                               kind   CHAR,
                               amount NUMBER) IS

/* Enters a bank transaction into the TRANSACTIONS table. A new
   transaction is always put into this 'queue' before being
   applied to the specified account by the APPLY_TRANSACTIONS
   procedure. Therefore, many transactions can be simultaneously
   input without interference. */

  BEGIN
        INSERT INTO transactions
          VALUES (acct, kind, amount, 'Pending', sysdate);
        COMMIT;
  END enter_transaction;

END bank_transactions;
```

Packages allow the database administrator or application developer to organize similar routines. They also offer increased functionality and database performance.

See ALso: Chapter 20, "Oracle Dependency Management"

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 might contain ten procedures. You can define the package so that only three procedures are public and therefore available for execution by a user of the package; the 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 26, "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 via 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.

Dependency Tracking for Packages

A package depends on the objects referenced by the procedures and functions defined in its body. Oracle automatically tracks and manages such dependencies.

See Also: Chapter 20, "Oracle Dependency Management"

Oracle Supplied Packages

Oracle supplies many PL/SQL packages that contain procedures for extending the functionality of the database or PL/SQL. Most of these packages have names that start with the DBMS_ prefix, such as DBMS_SQL, DBMS_LOCK, and DBMS_JOB. Some supplied packages have the UTL_ prefix, such as UTL_HTTP and UTL_FILE, or other prefixes including DEBUG_ and OUTLN_.

See Also: *Oracle8i Supplied PL/SQL Packages Reference* for detailed documentation of the Oracle supplied packages

How Oracle Stores Procedures and Packages

When you create a procedure or package, Oracle:

- Compiles the procedure or package
- Stores the compiled code in memory
- Stores the procedure or package in the database

Compiling Procedures and Packages

The PL/SQL compiler compiles the source code. The PL/SQL compiler is part of the PL/SQL engine contained in Oracle. If an error occurs during compilation, a message is returned. You can anticipate errors by using an error handler within the program, thus preventing Oracle from terminating the procedure.

See Also: *Oracle8i Application Developer's Guide - Fundamentals* for information about identifying compilation errors

Storing the Compiled Code in Memory

Oracle caches the compiled procedure or package in the shared pool of the system global area (SGA). This allows the code to be executed quickly and shared among many users. The compiled version of the procedure or package remains in the shared pool according to the modified least-recently-used algorithm used by the shared pool, even if the original caller of the procedure terminates his or her session.

See Also: "The Shared Pool" on page 7-7 for specific information about the shared pool buffer

Storing Procedures or Packages in Database

At creation and compile time, Oracle automatically stores the following information about stored procedures or packages in the database:

schema object name	This name identifies the procedure or package. You specify this name in the CREATE PROCEDURE, CREATE FUNCTION, CREATE PACKAGE, or CREATE PACKAGE BODY statement.
source code and parse tree	The PL/SQL compiler parses the source code and produces a parsed representation of the source code, called a *parse tree*.
pseudocode (P code)	The PL/SQL compiler generates the *pseudocode*, or P code, based on the parsed code. The PL/SQL engine executes this when the procedure or package is invoked.
error messages	Oracle might generate errors during the compilation of a procedure or package.

To avoid unnecessary recompilation of a procedure or package, both the parse tree and the P code of an object are stored in the database. This allows the PL/SQL engine to read the compiled version of a procedure or package into the shared pool buffer of the SGA when it is invoked and not currently in the SGA. The parse tree is used when the code calling the procedure is compiled.

All parts of database procedures are stored in the data dictionary (which is in the SYSTEM tablespace) of the corresponding database. When planning the size of the SYSTEM tablespace, the database administrator should keep in mind that all stored procedures require space in this tablespace.

How Oracle Executes Procedures and Packages

When you invoke a standalone or packaged procedure, Oracle performs the following tasks:

- Verifying User Access
- Verifying Procedure Validity
- Executing a Procedure

The verification and execution differ for definer-rights procedures and invoker-rights procedures.

See Also: "Dependency Tracking for Stored Procedures" on page 17-11

Verifying User Access

Oracle verifies that the calling user owns or has the EXECUTE privilege on the procedure or encapsulating package. The user who executes a procedure does not require access to any procedures or objects referenced within the procedure; only the creator of a procedure or package requires privileges to access referenced schema objects.

Verifying Procedure Validity

Oracle checks the data dictionary to determine whether the status of the procedure or package is valid or invalid. A procedure or package is invalid when one of the following has occurred since the procedure or package was last compiled:

- One or more of the schema objects referenced within the procedure or package, such as tables, views, and other procedures, have been altered or dropped. For example, a user added a column to a table.
- A system privilege that the package or procedure requires has been revoked from PUBLIC or from the owner of the procedure or package.

- A required schema object privilege for one or more of the schema objects referenced by a procedure or package has been revoked from PUBLIC or from the owner of the procedure or package.

A procedure is *valid* if it has not been invalidated by any of the above operations. If a valid standalone or packaged procedure is called, then the compiled code is executed. If an invalid standalone or packaged procedure is called, then it is automatically recompiled before being executed.

> **See Also:** Chapter 20, "Oracle Dependency Management" for a complete discussion of valid and invalid procedures and packages, recompiling procedures, and a thorough discussion of dependency issues

Executing a Procedure

The PL/SQL engine executes the procedure or package using different steps, depending on the situation:

- If the procedure is valid and currently in memory, the PL/SQL engine simply executes the P code.
- If the procedure is valid and currently not in memory, the PL/SQL engine loads the compiled P code from disk to memory and executes it. For packages, all constructs of the package (all procedures, variables, and so on, compiled as one executable piece of code) are loaded as a unit.

The PL/SQL engine processes a procedure statement by statement, handling all procedural statements by itself and passing SQL statements to the SQL statement executor, as illustrated in Figure 15–2 on page 15-17.

Name Resolution for Database Objects and Program Units

For a definer-rights procedure, all external references are resolved in the definer's schema. For an invoker-rights procedure, the resolution of external references depends on the kind of statement they appear in.

- External references in DML statements and dynamic SQL statements are resolved in the invoker's schema, and Oracle checks for access privileges at runtime using the invoker's rights. This rule applies to the names of database objects (such as tables, views, and sequences) in any of the following types of statements:
  - SELECT, UPDATE, INSERT, and DELETE statements

  - OPEN cursor (SELECT statements in a cursor declaration are resolved at OPEN)
  - LOCK TABLE statements
  - dynamic SQL statements: EXECUTE IMMEDIATE and PREPARE
  - DBMS_SQL statements: any statement parsed using DBMS_SQL.PARSE()

  Although the names of schema objects are resolved at run time, the compiler identifies a type for each such reference at compile time by temporarily resolving the name to a template object in the definer's schema.

- External references in all statements other than DML or dynamic SQL are resolved in the definer's schema, and Oracle checks for access privileges using the definer's rights at compile time. This rule applies to names of other program units called by the procedure (such as packages, procedures, functions, and types).

  For example, in an assignment statement involving an external function call "x := func(1)", the function name "func" resolves in the schema of the procedure's definer and the access to "func" is checked at compile time with the definer's privileges.

Name Resolution for Database Links

Database link names in PL/SQL procedures are resolved following the rules in the previous section. The authorization ID used to connect to the remote database is one of the following:

1. For fixed user links, Oracle uses the username specified in the link to connect to the remote database. This behavior is the same in definer-rights procedures and invoker-rights procedures.

   ```
   CREATE DATABASE LINK link1
       CONNECT TO scott IDENTIFIED BY tiger
       USING connect_string;
   ```

   If a procedure owned by JOE uses LINK1, no matter who invokes the procedure, the connection is as SCOTT because that is the name specified in the link.

2. For connected user links, Oracle uses the session username to connect to the remote database. This behavior is the same in definer-rights procedures and invoker-rights procedures.

   ```
   CREATE DATABASE LINK link2
   ```

```
USING connect_string;
```

If a procedure owned by JOE uses a connected user link LINK2, and a user SCOTT invokes the procedure, the connection to the remote database is as SCOTT.

3. For current user links, the behavior differs for definer-rights procedures and invoker-rights procedures:

   – For an invoker-rights procedure, Oracle uses the invoker's authorization ID to connect as a remote user.

   ```
   CREATE DATABASE LINK link3
       CONNECT TO CURRENT_USER
       USING connect_string;
   ```

   If a global user SCOTT invokes the invoker-rights procedure owned by JOE, then LINK3 connects to the remote database as user SCOTT because SCOTT is the current user.

   – For a definer-rights procedure, Oracle uses the owner's authorization ID to connect as a remote user. If the definer-rights procedure is owned by JOE, LINK3 connects to the remote database as global user JOE who then becomes the current user.

   See Also: *Oracle8i Distributed Database Systems*

18

Advanced Queuing

This chapter describes the Oracle Advanced Queuing feature. The chapter includes:

- Introduction to Message Queuing
- Oracle Advanced Queuing
  - Queuing Models
  - Queuing Entities
  - Features of Advanced Queuing

Note: The features described in this chapter are available only if you have purchased Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for information about the features available with Oracle8*i* Enterprise Edition.

See Also: *Oracle8i Application Developer's Guide - Advanced Queuing*

Introduction to 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 may 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 should implement content-based routing, content-based subscription, and content-based querying.

.A messageing 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.

Oracle Advanced Queuing

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 like transaction processing monitors or message-oriented middleware.

Oracle Advanced Queuing offers the following functionality:

- Point-to-point and publish-subscribe models of communication
- Structured payload for messages
- Priority and ordering of messages in queues
- Ability to specify a window of execution for each message
- Ability to query queues using standard SQL
- Integrated transactions to simplify application development and management
- Ability to dequeue multiple messages as a bundle
- Ability to specify multiple recipients
- Ability to propagate messages to queues in local or remote Oracle databases
- Rule- based publish-subscribe with content-based filtering
- Ability to wait for messages on multiple queues
- Persistent and non-persistent queuing
- Statistics on messages stored in queues and propagated to other queues
- Retention of messages and message history for analysis purposes
- Queue level access control
- Support for Oracle Parallel Server environment to achieve higher performance
- Asynchronous notification using callback functions

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

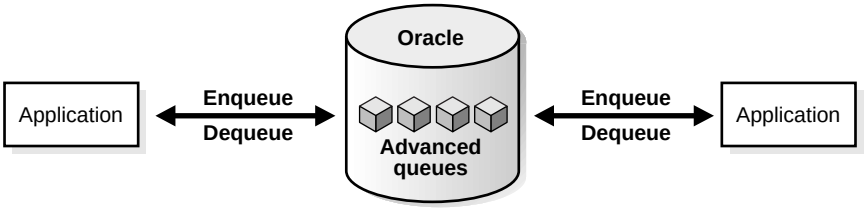
Applications can access the queuing functionality through the native interface for Oracle Advanced Queuing (defined in PL/SQL, C/C++, Java, and Visual Basic) or through the Java Messaging Service interface for Oracle Advanced Queuing (Java API based on the Java Messaging Service standard).

Queuing Models

Oracle Advanced Queuing can be set up according to the point-to-point model or the publish-subscribe model.

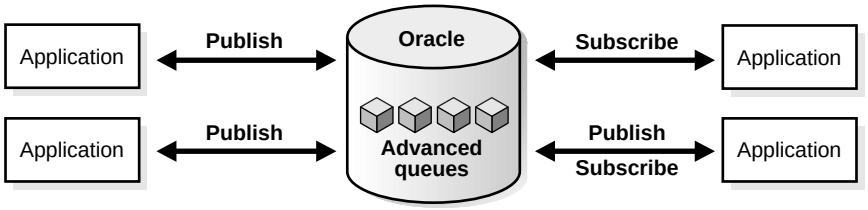
In the *point-to-point model*, senders and receivers decide upon a common queue in which to exchange messages. Each message is consumed by only one recipient. Figure 18–1, "Point-to-Point Model", shows that each application has its own message queue:

Figure 18–1 Point-to-Point Model



The *publish-subscribe model* is based on topics. Recipients subscribe to topics. Each message may be consumed by multiple recipients. Publishers are not always aware of subscribers. Figure 18–2, "Publish-Subscribe Model"shows that each application can publish or subscribe on a given topic:

Figure 18–2 Publish-Subscribe Model



Queuing Entities

Oracle Advanced Queuing has several basic entities:

- Messages
- Queues
- Queue Tables
- Agents
- Recipients
- Subscribers
- Recipient and Subscription Lists
- Rules
- Rule-Based Subscribers
- Queue Monitor

Messages

A *message* is the smallest unit of information being inserted into and retrieved from a queue. Messages generally encapsulate business data and events. A message consists of control information and payload data. The control information represents message properties used by Oracle Advanced Queuing to manage messages. The payload data is the information stored in the queue and is transparent to Oracle Advanced Queuing. The datatype of the payload can be either RAW or an object type.

A message can reside in only one queue. A message is created by the enqueue call and consumed by the dequeue call. Enqueue and dequeue calls are part of the DBMS_AQ package.

Queues

A *queue* is the repository for messages. There are two types of queues: user queues and exception queues. User queues are for normal message processing. There are two types of user queues: single-consumer queues and multi-consumer queues. *Single-consumer queues* are used in the point-to-point model. *Multi-consumer queues* are used in the publish-subscribe model. Messages are transferred from a user queue to an exception queue if they cannot be retrieved and processed for some reason.

All messages in a queue must have the same datatype.

Queues can be created, altered, started, stopped, and dropped by using the DBMS_AQADM package.

Queue Tables

Queues are stored in *queue tables*. Each queue table is a database table and contains one or more queues. Each queue table contains a default exception queue.

Creating a queue table creates a database table with approximately 25 columns. These columns store Oracle Advanced Queuing metadata and the user-defined payload.

A view and two indexes are created on the queue table. The view allows you to query the message data. The indexes are used to accelerate access to message data.

Agents

An *agent* is a queue user. There are two types of agents: producers who place messages in a queue (enqueuing) and consumers who retrieve messages

(dequeuing). Any number of producers and consumers can access the queue at a given time.

An agent is identified by its name, address, and protocol. For an agent on a remote database, the only protocol currently supported is an Oracle database link, using an address of the form *queue_name@dblink*.

Recipients

The *recipient* of a message may be specified by its name only, in which case the recipient must dequeue the message from the queue in which the message was enqueued. The recipient may be specified by name and an address with a protocol value of 0. The address should be the name of another queue in the same database or another Oracle8 database (identified by the database link), in which case the message is propagated to the specified queue and can be dequeued by a consumer with the specified name.

If the recipient's name is NULL, the message is propagated to the specified queue in the address and can be dequeued by the subscribers to the queue specified in the address. If the protocol field is nonzero, the name and address field is not interpreted by the system and the message can be dequeued by special consumer.

> **See Also:** "Propagating Messages to Other Databases" on page 18-12

Subscribers

A queue administrator can define a default list of recipients who can retrieve a message from a multi-consumer queue. These recipients are known as subscribers. Different queues can have different subscribers, and the same recipient can be a subscriber to more than one queue.

Recipient and Subscription Lists

A single message can be designed for consumption by multiple consumers. There are two ways to do this:

- The enqueuer can explicitly specify the consumers who may retrieve the message as recipients of the message. A recipient is an agent identified by a name, address and protocol.
- A queue administrator can specify a list of subscribers. If a message is enqueued without specifying the recipients, the message is implicitly sent to all the subscribers to that queue.

Specific messages in a queue can be directed toward specific recipients who may or may not be subscribers to the queue, thereby overriding the subscriber list.

Rules

A *rule* is used to define one or more subscribers' interest in subscribing to messages that conform to that rule. The messages that meet this criterion are then delivered to the interested subscribers. Put another way: a rule filters for messages in a queue on a subject in which a subscriber is interested.

A rule is specified as a boolean expression (one that evaluates to true or false) using syntax similar to the WHERE clause of a SQL query. This boolean expression can include conditions on:

- Message properties (`priority` and `corrid`)
- User data properties (object payloads only)
- Functions (as specified in the WHERE clause of a SQL query)

Rule-Based Subscribers

A *rule-based subscriber* is a subscriber that has a rule associated with it in the default recipient list. A rule-based subscriber is sent a message that has no explicit recipients specified if the associated rule evaluates to `TRUE` for the message.

Queue Monitor

The *queue monitor* is an optional background process that monitors messages in the queue. It provides the mechanism for message expiration, retry, and delay and allows you to collect interval statistics.

The queue monitor process is different from most other Oracle background processes in that process failure does not cause the instance to fail.

The initialization parameter AQ_TM_PROCESSES specifies creation of one or more queue monitor processes at instance startup.

See Also:

- "Windows of Execution" on page 18-9
- "Queuing Statistics" on page 18-13

Features of Advanced Queuing

This section describes the major features of Oracle Advanced Queuing.

Structured Payload

You can use object types to structure and manage the payload. (The RAW datatype can be used for unstructured payloads.)

Integrated Database Level Operational Support

Oracle Advanced Queuing stores messages in tables. All standard database features are supported.

SQL Access

Messages are stored as database records. You can use SQL to access the message properties, the message history, and the payload. All available SQL technology, such as indexes, can be used to optimize the access to messages.

The AQ_ADMINISTRATOR role provides access to information about queues.

> **See Also:** *Oracle8i Administrator's Guide* for information about roles

Windows of Execution

You can specify that the consumption of a message has to occur in a specific time window. A message can be marked as available for processing only after a specified time elapses (a delay time) and as having to be consumed before a specified time limit expires.

The AQ_TM_PROCESS initialization parameter enables time monitoring on queue messages, which is used for messages that specify delay and expiration properties. Time monitoring must also be enabled if you want to collect interval statistics.

If this parameter is set to 1, Oracle creates one *queue monitor process* (QMN0) as a background process to monitor the messages. If it is set to 2 through 10, Oracle creates the specified number of QMN*n* processes. If the parameter is not specified or is set to 0, then queue monitor processes are not created. The procedures in the DBMS_AQADM package for starting and stopping queue monitor operations are only valid if at least one queue monitor process was started with this parameter as part of instance startup.

See Also:

- "Queuing Statistics" on page 18-13
- *Oracle8i Reference* for information about the AQ_TM_PROCESS initialization parameter

Multiple Consumers per Message

A single message can be consumed by multiple consumers. When using multi-consumer queues, a single message can be consumed by multiple consumers.

Navigation

You have several options for selecting a message from a queue. You can select the first message or, once you have selected a message and established a consistent-read snapshot, you can retrieve the next message based on the current snapshot. You acquire a new consistent-read snapshot every time you select the first message from the queue.

You can also retrieve a specific message using the message's correlation identifier.

Priority and Ordering of Messages

You have three options for specifying the order in which messages are consumed: A sort order that specifies which properties are used to order all message in a queue, a priority that can be assigned to each message, and a sequence deviation that allows you to position a message in relation to other messages.

If several consumers act on the same queue, a consumer will get the first message that is available for immediate consumption. A message that is in the process of being consumed by another consumer will be skipped.

Modes of DEQUEUE

A DEQUEUE request can either browse or remove a message. If a message is browsed it remains available for further processing. If a message is removed, it is not available any more for DEQUEUE requests. Depending on the queue properties a removed message may be retained in the queue table.

Waiting for the Arrival of Messages

A DEQUEUE could be issued against an empty queue. You can specify if and for how long the request is allowed to wait for the arrival of a message.

Retries with Delays

A message has to be consumed exactly once. If an attempt to dequeue a message fails and the transaction is rolled back, the message will be made available for reprocessing after a user-specified delay elapses. Reprocessing will be attempted up to the specified limit.

Exception Queues

A message may not be consumed within the given constraints, that is, within the window of execution or within the limits of the retries. If such a condition arises, the message will be moved to a user-specified exception queue.

Visibility

ENQUEUE/DEQUEUE requests are normally part of a transaction that contains the requests. This provides the desired transactional behavior. However, you can specify that a request is a transaction by itself, making the result of that request immediately visible to other transactions.

Message Grouping

Messages belonging to one queue can be grouped to form a set that can only be consumed by one user at a time. This requires that the queue be created in a queue table that is enabled for message grouping.

All messages belonging to a group have to be created in the same transaction and all messages created in one transaction belong to the same group. This feature allows you to segment complex messages into simple messages, for example, messages directed to a queue containing invoices could be constructed as a group of messages starting with the header message, followed by messages representing details, followed by the trailer message.

Retention

You can specify that messages be retained after consumption. This allows you to keep a history of relevant messages. The history can be used for tracking, data warehouse, and data mining operations.

Message History

Oracle Advanced Queuing stores information about the history of each message. The information contains the ENQUEUE/DEQUEUE time and the identification of the transaction that executed each request.

Tracking

If messages are retained they can be related to each other; for example, if a message *m2* is produced as a result of the consumption of message *m1*, *m1* is related to *m2*. This allows you to track sequences of related messages. These sequences represent *event journals* which are often constructed by applications. Oracle Advanced Queuing is designed to let applications create event journals automatically.

Queue Level Access Control

With Oracle 8*i*, an owner of an 8.1 style queue can grant or revoke queue level privileges on the queue. DBAs can grant or revoke new AQ system level privileges to any database user. DBAs can also make any database user an AQ administrator.

Propagating Messages to Other Databases

Messages enqueued in one database can be propagated to queues on another database. The datatypes of the source and destination queues must match each other.

Message propagation enables applications to communicate with each other without being connected to the same database or the same queue. Propagation uses database links and Net8 between local or remote databases, both of which must have Oracle Advanced Queuing enabled.

You can schedule or unschedule message propagation and specify the start time, the propagation window, and a date function for later propagation windows in periodic schedules. The data dictionary view DBA_QUEUE_SCHEDULES describes the current schedules for propagating messages.

The job queue background processes (SNP*n*) handle message propagation. To enable propagation, you must start at least one job queue process with the initialization parameter JOB_QUEUE_PROCESSES.

Propagation Statistics

Propagation statistics are available in the form of total/average number of messages/bytes propagated in a schedule. This information can be used to tune the performance of propagation of messages.

Non-Persistent Queues

AQ can deliver non-persistent messages asynchronously to subscribers. These messages can be event-driven and do not persist beyond the failure of the system

(or instance). AQ supports persistent and non-persistent messages with a common API.

Publish-Subscribe Support

A combination of features are introduced to allow a publish-subscribe style of messaging between applications. These features include rule-based subscribers, message propagation, the listen feature and notification capabilities. Triggers can be used to publish system events and user events.

> **See Also:** "Triggers on System Events and User Events" on page 19-19

Support for Oracle Parallel Server Environments

An application can specify the instance affinity for a queue-table. When AQ is used with parallel server and multiple instances, this information is used to partition the queue-tables between instances for queue-monitor scheduling. The queue-table is monitored by the queue-monitors of the instance specified by the user. If an instance affinity is not specified, the queue-tables will be arbitrarily partitioned among the available instances. There can be pinging between the application accessing the queue-table and the queue-monitor monitoring it. Specifying the instance-affinity does not prevent the application from accessing the queue-table and its queues from other instances.

This feature prevents pinging between queue monitors and AQ propagation jobs running in different instances. In Oracle8*i* release 8.1.5 an instance affinity (primary and secondary) can be specified for a queue table. When AQ is used with parallel server and multiple instances, this information is used to partition the queue-tables between instances for queue-monitor scheduling as well as for propagation. At any time, the queue table is affiliated to one instance. In the absence of an explicitly specified affinity, any available instance is made the owner of the queue table. If the owner of the queue table dies, the secondary instance or some available instance takes over the ownership for the queue table.

Queuing Statistics

Oracle Advanced Queuing keeps statistics about the current state of the queuing system as well as time-interval statistics in the dynamic table GV$AQ.

Statistics about the current state of the queuing system include the numbers of ready, waiting, and expired messages.

One or more queue monitor processes must be started to keep interval statistics, which include:

- The number of messages in each state (ready, waiting, and expired)
- The average wait time of waiting messages
- The total wait time of waiting messages

Asynchronous Notification

Instead of using blocking dequeue calls, clients may choose to receive notifications when a message matching the client's selection criteria appears in the queue.

Java Messaging Service clients can register MessageListeners to receive messages asynchronously. Oracle Advanced Queuing invokes the onMessage callback when a message for the client is received.

OCI clients can use the new call OCISubscriptionRegister to register a callback for message notification. The client issues a registration call which specifies a subscription name and a callback. When messages for the subscription are received, the callback is invoked. The callback may then issue an explicit dequeue to retrieve the message.

Listen Capability (Wait for Messages on Multiple Queues)

The listen call can be used to wait for messages on multiple queues. It can be used by a gateway application to monitor a set of queues. An application can also use it to wait for messages on a list of subscriptions. In the native interface for Oracle Advanced Queuing (PL/SQL or OCI), listen is a blocking call. If the listen returns successfully, a dequeue must be used to retrieve the message.

In the Java Messaging Service interface, Session Message Listener can be used to listen for messages on multiple queues. This is nonblocking. When a message shows up for any of the consumers in a session, the onMessage callback of the MessageListener is invoked.

Correlation Identifier

You can assign an identifier to each message. This identifier can be used to retrieve specific messages.

Import/Export

The import/export of queues constitutes the import/export of the underlying queue tables and related dictionary tables. Import and export of queues can only be done at queue table granularity.

When a queue table is exported, both the table definition information and the queue data are exported. When a queue table is imported, export action procedures maintain the queue dictionary. Because the queue table data is also exported, the user is responsible for maintaining application-level data integrity when queue table data are being transported.

For every queue table that supports multiple recipients, there is an index-organized table that contains important queue metadata. This metadata is essential to the operations of the queue, so you must export and import this index-organized table as well as the queue table for the queues in this table to work after import.

When the schema containing the queue table is exported, the index-organized table is also automatically exported. The behavior is similar at import time. Because the metadata table contains rowids of some rows in the queue table, import issues a note about the rowids being obsolete when importing the metadata table. This message can be ignored, as the queuing system automatically corrects the obsolete rowids as a part of the import process. However, if another problem is encountered while doing the import (such as running out of rollback segment space), the problem should be corrected and the import should be rerun.

See Also:

- *Oracle8i Application Developer's Guide - Advanced Queuing*
- *Oracle8i Utilities*

19

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:

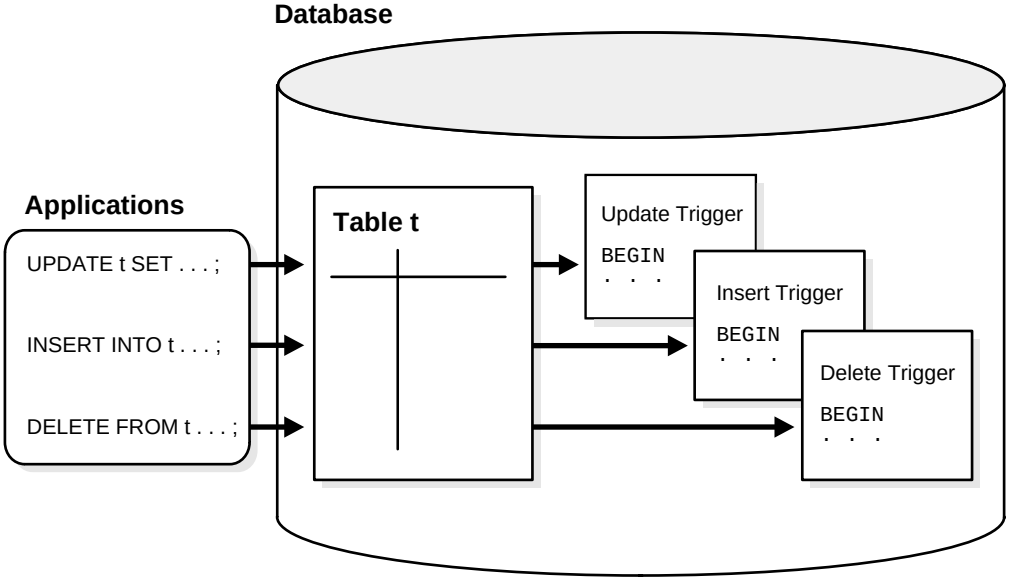
Introduction to Triggers

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 19–1 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 19–1 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 17, "Procedures and Packages" for information on the similarities of triggers to stored procedures
- "Triggering Event or Statement" on page 19-7

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. A trigger can also restrict DML operations to occur only at certain times during weekdays. 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 sophisticated auditing
- Maintain synchronous table replicates

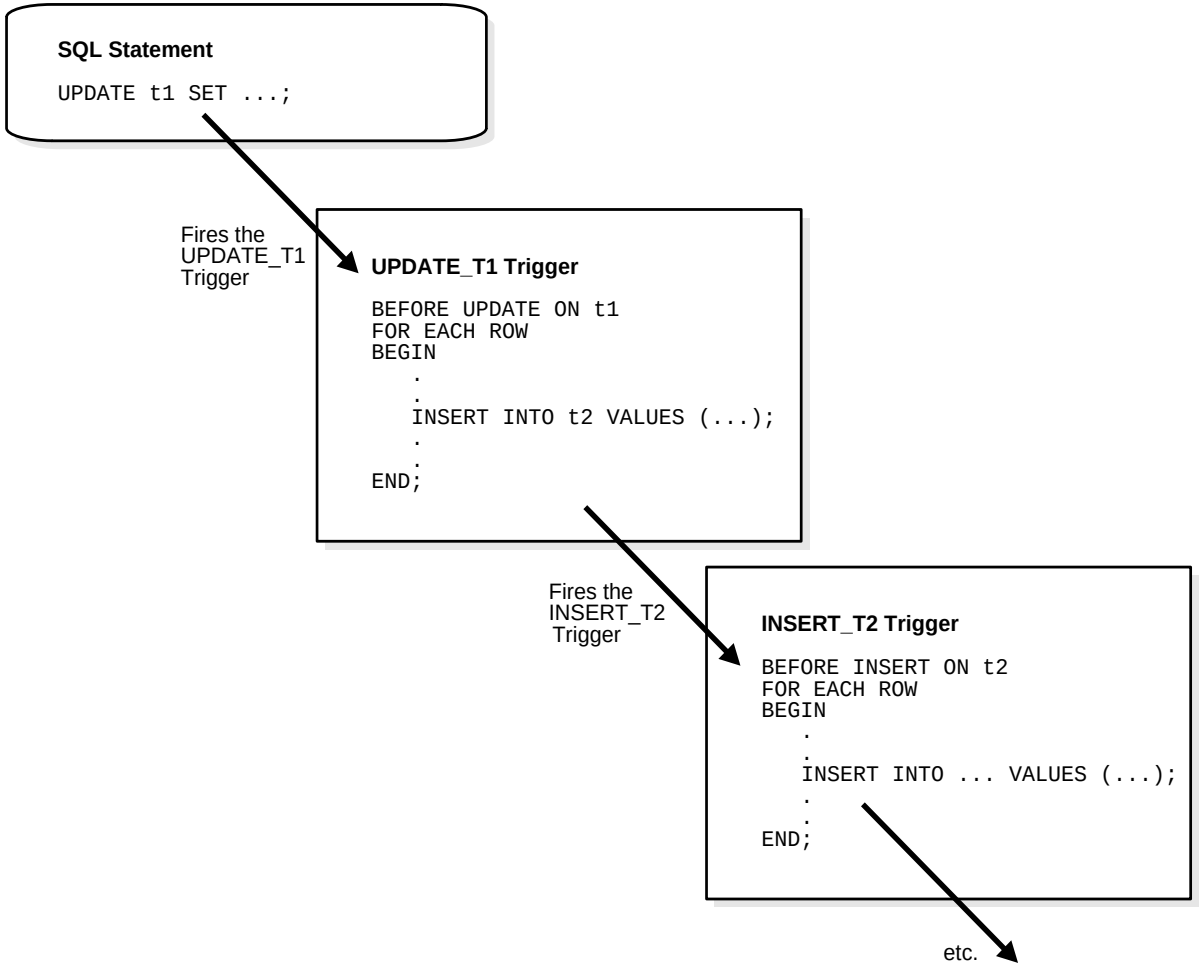
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Publish information about database events, user events, and SQL statements to subscribing applications

> **See Also:** *Oracle8i 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 may 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 19–2 illustrates cascading triggers.

Figure 19–2 Cascading Triggers



Triggers versus 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 key
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK
  - DELETE CASCADE
  - DELETE SET NULL

  See Also: "How Oracle Enforces Data Integrity" on page 25-4 for more information about integrity constraints

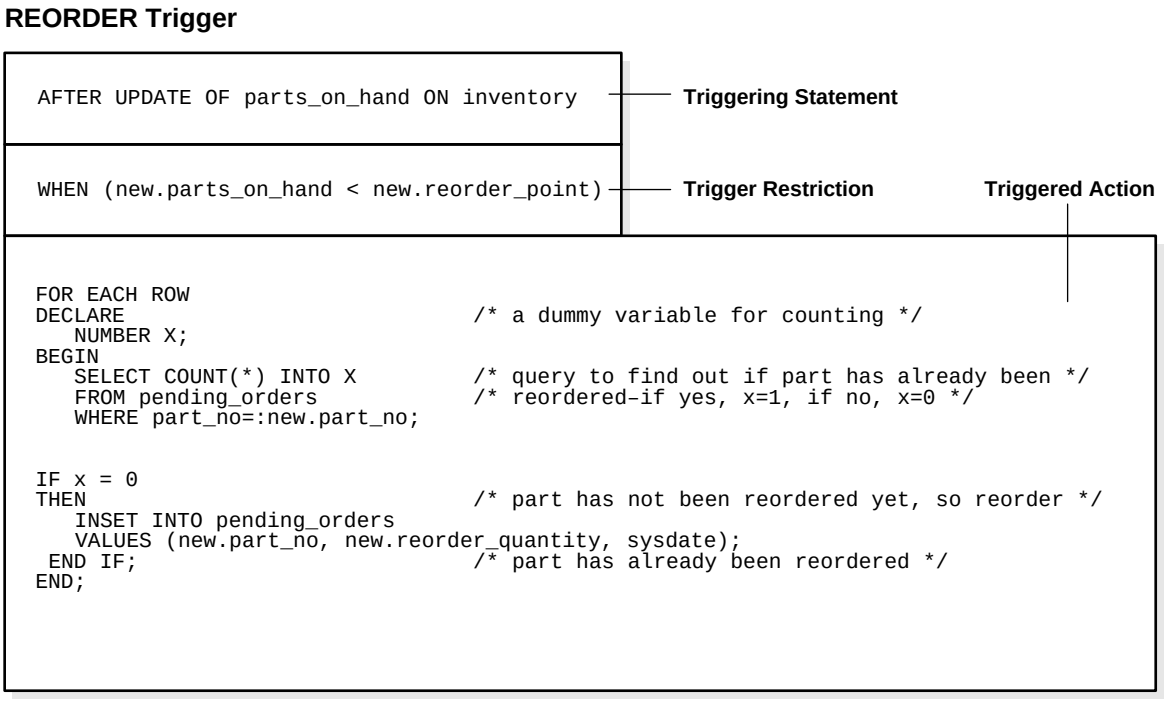
Parts of a Trigger

A trigger has three basic parts:

- A triggering event or statement
- A trigger restriction
- A trigger action

Figure 19–3 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 19–3 The REORDER Trigger



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 19–3, 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 19–3 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 19-13
- "Triggers on System Events and User Events" on page 19-19 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 should determine 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 above, 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 may 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 snapshot 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 example below 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

> **Note:** INSTEAD OF triggers are available only if you have purchased the Oracle8*i* Enterprise Edition. They can be used with relational views and object views. See *Getting to Know Oracle8i* for information about the features available in Oracle8*i* Enterprise Edition.

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.

Modifying 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 will no longer be 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 will be 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 (see the next section). 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 would perform 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 14, "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 below. 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 10-16

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');
```

will fire the `manager_info_insert` trigger and update the underlying tables. Similar triggers can specify appropriate actions for INSERT and DELETE on the view.

Usage Notes

The INSTEAD OF clause to the CREATE TRIGGER statement can *only* be used for triggers created over views. The BEFORE and AFTER clauses *cannot* be used for triggers created over views.

The CHECK clause for views is not enforced when inserts or updates to the view are done using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals*
- CREATE TRIGGER statement in *Oracle8i SQL Reference*

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.

Consider the department employee example again. Let the department view contain the list of departments and the set of employees in each department. The

following example shows how to modify the elements of the nested table of employee objects in the department view using INSTEAD OF triggers.

```
/* Create an employee type */
CREATE TYPE emp_t AS OBJECT
(
  empno NUMBER,
  empname VARCHAR2(20),
  days_worked NUMBER
);
/
/* Create a nested table type of employees */
CREATE TYPE emplist_t AS TABLE OF emp_t;
/
/* Now, create the department type */
CREATE TYPE dept_t AS OBJECT
(
  deptno NUMBER,
  deptname VARCHAR2(20),
  emplist  emplist_t
);
/
/* The dept_view can now be created based on the dept (department) and emp
 * (employee) tables. */
CREATE VIEW dept_view OF dept_t WITH OBJECT OID(deptno)
  AS SELECT d.deptno, d.deptname, -- department number and name
       CAST (MULTISET (
                SELECT e.empno, e.empname, (SYSDATE - e.startdate)
                FROM   emp e
                WHERE  e.deptno = d.deptno)
             AS emplist_t)       -- emplist - nested table of employees
     FROM dept d;
```

To be able to insert into the nested table `emplist` in the view using the TABLE syntax:

```
INSERT INTO TABLE
   (SELECT d.emplist FROM dept_view d WHERE d.deptno = 10)
   VALUES (10,'Harry Mart',334);
```

You can define an INSTEAD OF trigger on the nested table `emplist` to handle the insert:

```
CREATE TRIGGER dept_empinstr INSTEAD OF INSERT ON
   NESTED TABLE emplist OF dept_view FOR EACH ROW
BEGIN
```

```
    INSERT INTO emp VALUES(:NEW.empno, :NEW.empname,
                           :PARENT.deptno, SYSDATE - :NEW.days_worked);
END;
/
```

Similarly you can define triggers to handle updates and deletes on the nested table elements.

Accessing the Attributes of the Parent Row Inside a Nested Table Trigger In regular triggers, the current row's values can be accessed using the NEW and OLD qualifiers. For triggers on nested table columns of views, these qualifiers refer to the attributes of the nested table element being modified. In order to access the values of the parent row containing this nested table column, you can use the PARENT qualifier.

This qualifier can be used only inside these nested table triggers. The parent row's values obtained using this PARENT qualifier cannot be modified (that is, they are read-only).

Consider the `dept_empinstr` trigger example shown above. The NEW qualifier refers to the row of the nested table being inserted (that is, it contains `empno`, `empname` and `days_worked`) and does not include the department number (`deptno`) where the employee works. But you need to insert the department number into the employee table inside the trigger. This `deptno` value can be obtained from the parent row that contains the list of employees, using the PARENT qualifier.

Firing of Nested Table and View Triggers As explained before, if a nested table column in a view has an INSTEAD OF trigger defined over it, then when an element of that nested table is inserted, updated, or deleted, the trigger is fired to do the actual modification.

The view containing the nested table column need not have any INSTEAD OF triggers defined over it for this to work. Any triggers defined on the view will not fire for any modifications to the nested table elements.

Conversely, a statement that modifies a row in the view would only fire the triggers defined on the view and not those on the nested table columns of that view. For instance if the `emplist` nested table column is updated through the `dept_view` as in:

```
UPDATE dept_view SET emplist = emplist_t(emp_t(1001,'John',234));
```

it will fire the INSTEAD OF update triggers defined over the `dept_view`, if any, but not the `dept_empinstr` nested table trigger.

See Also: *Oracle8i Application Developer's Guide - Fundamentals*

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: Chapter 18, "Advanced Queuing"

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 may not 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 may 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 19-9
- "BEFORE and AFTER Triggers" on page 19-10
- *Oracle8i 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:

1. Original SQL statement issued.

    2. BEFORE row triggers fired.

        3. AFTER statement triggers fired by UPDATE in BEFORE row trigger.

            4. Statements of AFTER statement triggers executed.

            5. Integrity constraint checked on tables changed by AFTER statement triggers.

        6. Statements of BEFORE row triggers executed.

        7. Integrity constraint checked on tables changed by BEFORE row triggers.

    8. SQL statement executed.

9. 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 snapshot 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.

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.

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 (in other words, 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:** "How Oracle Stores Procedures and Packages" on page 17-17 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:** '"How Oracle Executes Procedures and Packages" on page 17-19

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 20, "Oracle Dependency Management"

20

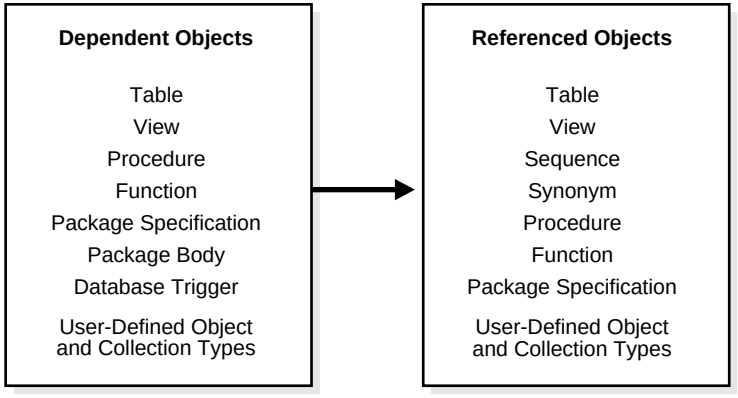
Oracle Dependency Management

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

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 20–1 illustrates the different types of dependent and referenced objects:

Figure 20–1 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.

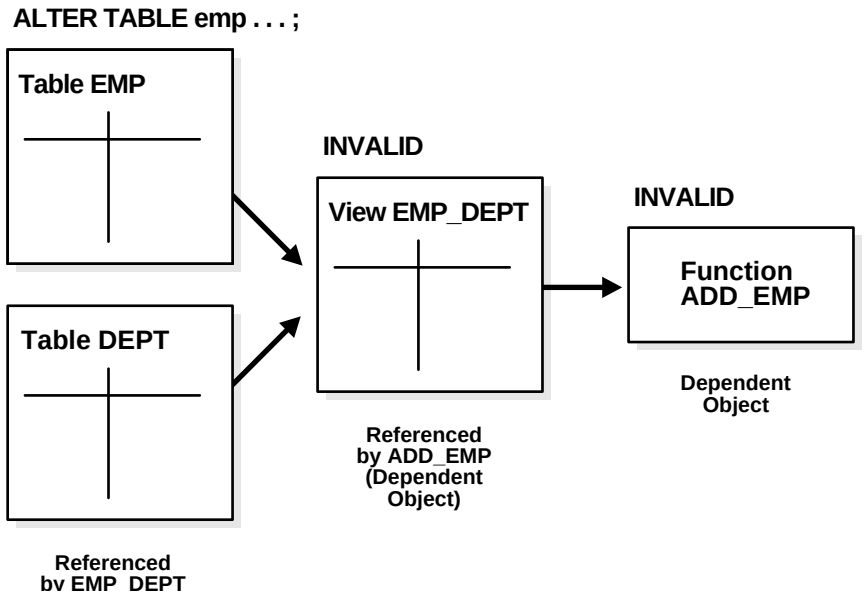
If a view, procedure, function, or package is invalid, Oracle may have attempted to compile it, but errors relating to the object occurred. For example, when compiling a view, one of its base tables might not exist, or the correct privileges for the base table might not be present. When compiling a package, 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 not only changes the status 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 20–2 illustrates indirect dependencies:

Figure 20–2 Indirect Dependencies



Resolving 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.

Compiling 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.

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 last case reveals a mechanism for minimizing dependencies among procedures and referenced objects by using packages.

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 timestamp.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for information on 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, since NULL values are not stored in indexes.

The following sections describe additional requirements.

> **See Also:** "Function-Based Indexes" on page 10-26

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: *Oracle8i Designing and Tuning for Performance*

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.

Resolving 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 example above, 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.

Managing 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.

Managing 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 *timestamp checking* or *signature checking*.

The dynamic initialization parameter REMOTE_DEPENDENCIES_MODE determines whether timestamps or signatures govern remote dependencies.

> **See Also:** *Oracle8i Application Developer's Guide - Fundamentals* for details about managing remote dependencies withtimestamps or signatures

Timestamp Checking In the timestamp checking dependency model, whenever a procedure is compiled or recompiled its *timestamp* (the time it is created, altered, or replaced) is recorded in the data dictionary. The timestamp 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 timestamp.

When a dependent procedure is used, Oracle compares the remote timestamps recorded at compile time with the current timestamps 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 timestamps match.
- The local procedure is invalidated if any timestamps 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 timestamp 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 timestamp of the remote procedure, every local procedure that depends on the remote procedure is invalidated.

Actual timestamp comparison occurs when a statement in the body of a local procedure executes a remote procedure; only at this moment are the timestamps compared via 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 thefollowing 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

Optimization of SQL Statements

Part VI describes the optimizer, which chooses the most efficient way to execute each SQL statement.

Part VI contains the following chapter:

- Chapter 21, "The Optimizer"

21

The Optimizer

This chapter introduces the Oracle optimizer. It includes:

- Introduction to Optimization
- Cost-Based Optimization
- Extensible Optimization
- Rule-Based Optimization

> **See Also:** *Oracle8i Designing and Tuning for Performance* for more information about the optimizer, including how to use materialized views for query rewrites

Introduction to Optimization

Optimization is the process of choosing 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, or DELETE. Many different ways to execute a SQL statement often exist, 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* calculates the most efficient way to execute a SQL statement. The optimizer evaluates many factors to select among alternative access paths. It can use a cost-based or rule-based approach.

Note: The optimizer may not make the same decisions from one version of Oracle to the next. In more recent versions, the optimizer may make different decisions based on better, more sophisticated information available to it.

You can influence the optimizer's choices by setting the optimizer approach and goal and by gathering statistics for cost-based optimization. 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:

- "Cost-Based Optimization" on page 21-8
- "Rule-Based Optimization" on page 21-20
- *Oracle8i Designing and Tuning for Performance* for information about using hints in SQL statements

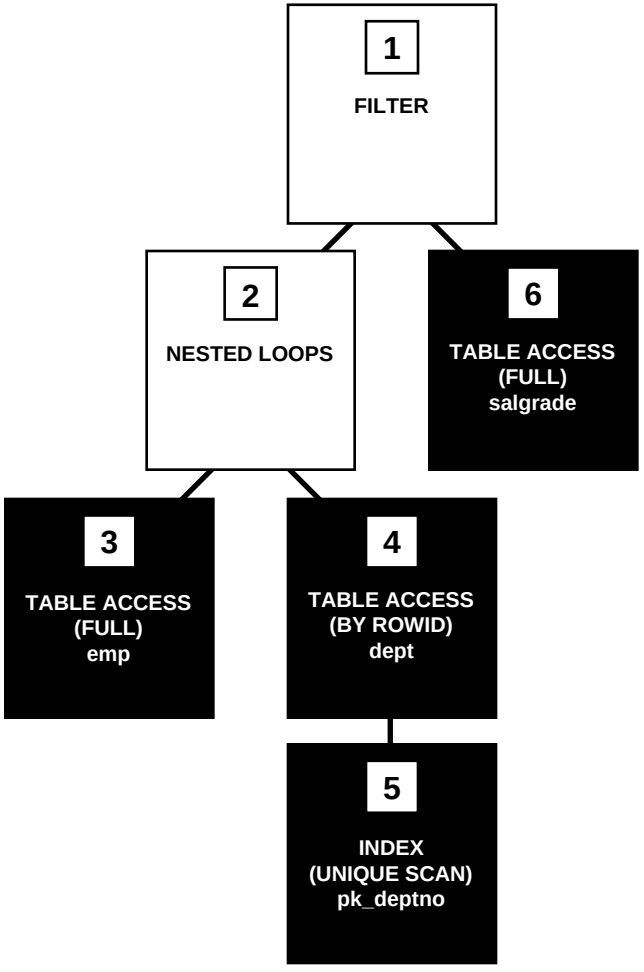
Execution Plans

To execute a DML statement, Oracle may have 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 the *join order* of the tables.

Figure 21–1 shows a graphical representation of the execution plan for the following SQL statement, which selects the name, job, salary, and department name for all employees whose salaries do not fall into a recommended salary range:

```
SELECT ename, job, sal, dname
    FROM emp, dept
  WHERE emp.deptno = dept.deptno
    AND NOT EXISTS
      (SELECT *
           FROM salgrade
         WHERE emp.sal BETWEEN losal AND hisal);
```

Figure 21–1 An Execution Plan



Steps of Execution Plan

Each step of the execution plan returns a set of rows that either are used by the next step or, in the last step, are returned to the user or application issuing the SQL statement. A set of rows returned by a step is called a *row source*.

Figure 21–1 is a hierarchical diagram showing the flow of row sources from one step to another. The numbering of the steps reflects the order in which they are

displayed in response to the EXPLAIN PLAN statement (described in the next section). This generally is **not** the order in which the steps are executed. Each step of the execution plan either retrieves rows from the database or accepts rows from one or more row sources as input:

- Steps indicated by the shaded boxes physically retrieve data from an object in the database. Such steps are called *access paths*:
  - Steps 3 and 6 read all the rows of the EMP and SALGRADE tables, respectively.
  - Step 5 looks up in the PK_DEPTNO index each DEPTNO value returned by Step 3. There it finds the rowids of the associated rows in the DEPT table.
  - Step 4 retrieves from the DEPT table the rows whose rowids were returned by Step 5.
- Steps indicated by the clear boxes operate on row sources:
  - Step 2 performs a nested loops operation, accepting row sources from Steps 3 and 4, joining each row from Step 3 source to its corresponding row in Step 4, and returning the resulting rows to Step 1.
  - Step 1 performs a filter operation. It accepts row sources from Steps 2 and 6, eliminates rows from Step 2 that have a corresponding row in Step 6, and returns the remaining rows from Step 2 to the user or application issuing the statement.

See Also:

- "Execution Order" on page 21-6
- *Oracle8i Designing and Tuning for Performance* for more information on access paths and methods by which Oracle joins row sources

The EXPLAIN PLAN statement

You can examine the execution plan chosen by the optimizer for a SQL statement by using the EXPLAIN PLAN statement, which causes the optimizer to choose the execution plan and then inserts data describing the plan into a database table.

For example, the following output table is such a description for the statement examined in the previous section:

```
ID    OPERATION           OPTIONS        OBJECT_NAME
------------------------------------------------------------
0     SELECT STATEMENT
```

```
1       FILTER
2         NESTED LOOPS
3           TABLE ACCESS  FULL           EMP
4           TABLE ACCESS  BY ROWID       DEPT
5             INDEX       UNIQUE SCAN    PK_DEPTNO
6           TABLE ACCESS  FULL           SALGRADE
```

Each box in Figure 21–1 and each row in the output table corresponds to a single step in the execution plan. For each row in the listing, the value in the ID column is the value shown in the corresponding box in Figure 21–1.

You can obtain such a listing by using the EXPLAIN PLAN statement and then querying the output table.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for information on how to use EXPLAIN PLAN

Execution Order

The steps of the execution plan are not performed in the order in which they are numbered. Rather, Oracle first performs the steps that appear as leaf nodes in the tree-structured graphical representation of the execution plan (Steps 3, 5, and 6 in Figure 21–1). The rows returned by each step become the row sources of its parent step. Then Oracle performs the parent steps.

To execute the statement for Figure 21–1, for example, Oracle performs the steps in this order:

- First, Oracle performs Step 3, and returns the resulting rows, one by one, to Step 2.
- For each row returned by Step 3, Oracle performs these steps:
  1. Oracle performs Step 5 and returns the resulting rowid to Step 4.
  2. Oracle performs Step 4 and returns the resulting row to Step 2.
  3. Oracle performs Step 2, joining the single row from Step 3 with a single row from Step 4, and returning a single row to Step 1.
  4. Oracle performs Step 6 and returns the resulting row, if any, to Step 1.
  5. Oracle performs Step 1. If a row is not returned from Step 6, Oracle returns the row from Step 2 to the user issuing the SQL statement.

Note that Oracle performs Steps 5, 4, 2, 6, and 1 once for each row returned by Step 3. If a parent step requires only a single row from its child step before it can be

executed, Oracle performs the parent step (and possibly the rest of the execution plan) as soon as a single row has been returned from the child step. If the parent of that parent step also can be activated by the return of a single row, then it is executed as well.

Thus the execution can cascade up the tree. Oracle performs the parent step and all cascaded steps once for each row in turn retrieved by the child step. The parent steps that are triggered for each row returned by a child step include table accesses, index accesses, nested loops joins, and filters.

If a parent step requires all rows from its child step before it can be executed, Oracle cannot perform the parent step until all rows have been returned from the child step. Such parent steps include sorts, sort-merge joins, and aggregate functions.

Optimizer Plan Stability

After carefully tuning an application, you might want to ensure that the optimizer generates the same execution plan whenever the same SQL statements are executed. *Plan stability* allows you to maintain the same execution plans for the same SQL statements, regardless of changes to the database. Changes to the database include:

- Re-analyzing tables
- Adding or deleting data
- Modifying a table's columns, constraints, or indexes
- Changing the system configuration
- Upgrading to a new version of the optimizer

The CREATE OUTLINE statement creates a *stored outline*, which contains a set of attributes that the optimizer uses to create an execution plan. Stored outlines can also be created automatically by setting the system parameter CREATE_STORED_OUTLINES to TRUE.

The system parameter USE_STORED_OUTLINES can be set to TRUE, FALSE, or a category name to indicate whether to make use of existing stored outlines for queries that are being executed. The OUTLN_PKG package provides procedures used for managing stored outlines.

Implementing plan stability creates a new schema called OUTLN, which is created with DBA privileges. The database administrator should change the password for the OUTLN schema just as for the SYS and SYSTEM schemas.

See Also:

- *Oracle8i Designing and Tuning for Performance* for information about using plan stability and for information about the CREATE OUTLINE statement
- *Oracle8i Supplied PL/SQL Packages Reference* for information about the OUTLN_PKG package

Cost-Based Optimization

Using the cost-based approach, the optimizer determines which execution plan is most efficient by considering available access paths and factoring in information based on statistics for the schema objects (tables or indexes) accessed by the SQL statement. The cost-based approach also considers hints, which are optimization suggestions placed in a Comment in the statement.

Conceptually, the cost-based approach consists of these steps:

1. The optimizer generates a set of potential execution plans for the SQL statement based on its available access paths and hints.
2. The optimizer estimates the cost of each execution plan based on statistics in the data dictionary for the data distribution and storage characteristics of the tables, indexes, and partitions accessed by the statement.

   The *cost* is an estimated value proportional to the expected resource use needed to execute the statement with a particular execution plan. The optimizer calculates the cost of each possible access method and join order based on the estimated computer resources, including (but not limited to) I/O, CPU time, and memory, that are required to execute the statement using the plan.

   Serial execution plans with greater costs take more time to execute than those with smaller costs. When using a parallel execution plan, however, resource use is not directly related to elapsed time.
3. The optimizer compares the costs of the execution plans and chooses the one with the smallest cost.

Goal of the Cost-Based Approach

By default, the goal of the cost-based approach is the best *throughput*, or minimal resource use necessary to process all rows accessed by the statement.

Oracle can also optimize a statement with the goal of best *response time*, or minimal resource use necessary to process the first row accessed by a SQL statement.

For parallel execution of a SQL statement, the optimizer can choose to minimize elapsed time at the expense of resource consumption. The initialization parameter OPTIMIZER_PERCENT_PARALLEL specifies how much the optimizer attempts to parallelize execution.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for information about using the OPTIMIZER_PERCENT_PARALLEL parameter and for information about how the optimizer chooses an optimization approach and goal

Statistics for Cost-Based Optimization

The cost-based approach uses statistics to calculate the selectivity of predicates and estimate the cost of each execution plan. *Selectivity* is the fraction of rows in a table that the SQL statement's predicate chooses. The optimizer uses the selectivity of a predicate to estimate the cost of a particular access method and to determine the optimal join order.

Statistics quantify the data distribution and storage characteristics of tables, columns, indexes, and partitions. The optimizer uses these statistics to estimate how much I/O, CPU time, and memory are required to execute a SQL statement using a particular execution plan. The statistics are stored in the data dictionary, and they can be exported from one database and imported into another. For example, you can transfer production statistics to a test system to simulate the real environment, even though the test system may only have small samples of data.

You must gather statistics on a regular basis to provide the optimizer with information about schema objects. New statistics should be gathered after a schema object's data or structure are modified in ways that make the previous statistics inaccurate. For example, after loading a significant number of rows into a table, you should collect new statistics on the number of rows. After updating data in a table, you do not need to collect new statistics on the number of rows but you might need new statistics on the average row length.

> **See Also:** "Gathering Statistics" on page 21-13

Histograms for Cost-Based Optimization

Cost-based optimization uses data value histograms to get accurate estimates of the distribution of column data. A *histogram* partitions the values in the column into bands, so that all column values in a band fall within the same range. Histograms

provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with nonuniform data distributions. Ranges in a histogram are called *buckets*.

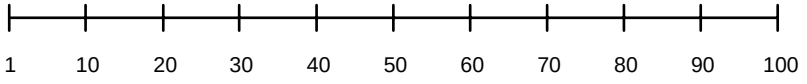
One of the fundamental capabilities of cost-based optimization is determining the selectivity of predicates that appear in queries. Selectivity estimates are used to decide when to use an index and the order in which to join tables. Most attribute domains (a table's columns) are *not* uniformly distributed.

Cost-based optimization uses *height-balanced histograms* on specified attributes to describe the distributions of nonuniform domains. In a height-balanced histogram, the column values are divided into bands so that each band contains approximately the same number of values. The useful information that the histogram provides, then, is where in the range of values the endpoints fall.

Comparing Height-balanced and Width-balanced Histograms Oracle uses *height-balanced* histograms as opposed to *width-balanced* histograms. The two types of histograms differ in the following ways:

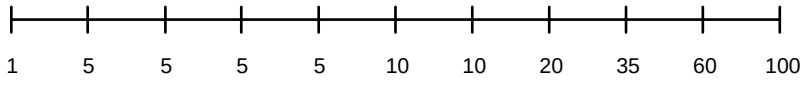
- Width-balanced histograms divide the data into a fixed number of equal-width ranges and then count the number of values falling into each range.
- Height-balanced histograms place approximately the same number of values into each range so that the endpoints of the range are determined by how many values are in that range.

Consider a column C with values between 1 and 100 and a histogram with 10 buckets. If the data in C is uniformly distributed, this histogram would look like this, where the numbers are the endpoint values:



The number of rows in each bucket is one tenth the total number of rows in the table. Four-tenths of the rows have values between 60 and 100 in this example of uniform distribution.

If the data is not uniformly distributed, the histogram might look like this:



In this case, most of the rows have the value 5 for the column. In this example, only 1/10 of the rows have values between 60 and 100.

Suppose that the values in a single column of a 1000-row table range between 1 and 100, and suppose that you want a 10-bucket histogram. In a width-balanced histogram, the buckets would be of equal width (1-10, 11-20, 21-30, and so on) and each bucket would count the number of rows that fall into that bucket's range. In a height-balanced histogram, each bucket has the same height (in this case 100 rows) and the endpoints for each bucket are determined by the density of the distinct values in the column.

The advantage of the height-balanced approach is clear when the data is highly skewed. Suppose that 800 rows of a 1000-row table have the value 5, and the remaining 200 rows are evenly distributed between 1 and 100. A width-balanced histogram would have 820 rows in the bucket labeled 1-10 and approximately 20 rows in each of the other buckets. The height-based histogram would have one bucket labeled 1-5, seven buckets labeled 5-5, one bucket labeled 5-50, and one bucket labeled 50-100.

If you want to know how many rows in the table contain the value 5, the height-balanced histogram shows that approximately 80% of the rows contain this value. However, the width-balanced histogram does not provide a mechanism for differentiating between the value 5 and the value 6. You would compute only 8% of the rows contain the value 5 in a width-balanced histogram. Therefore height-based histograms are more appropriate for determining the selectivity of column values.

When to Use Histograms Histograms can affect performance and should be used only when they substantially improve query plans. In general, you should create histograms on columns that are frequently used in WHERE clauses of queries and have a highly skewed data distribution. For many applications, it is appropriate to create histograms for all indexed columns because indexed columns typically are the columns most often used in WHERE clauses.

Histograms are persistent objects, so there is a maintenance and space cost for using them. You should compute histograms only for columns that you know have highly

skewed data distribution. For uniformly distributed data, cost-based optimization can make fairly accurate guesses about the cost of executing a particular statement without the use of histograms.

Histograms, like all other optimizer statistics, are static. They are useful only when they reflect the current data distribution of a given column. The data in the column can change as long as the *distribution* remains constant. If the data distribution of a column changes frequently, you must recompute its histogram frequently.

Histograms are *not* useful for columns with the following characteristics:

- All predicates on the column use bind variables.
- The column data is uniformly distributed.
- The column is not used in WHERE clauses of queries.
- The column is unique and is used only with equality predicates.

You generate histograms by using the DBMS_STATS package or the ANALYZE statement. You can generate histograms for columns of a table or partition. Histogram statistics are not collected in parallel.

You can view histogram information with the following data dictionary views:

- USER_HISTOGRAMS, ALL_HISTOGRAMS, and DBA_HISTOGRAMS
- USER_PART_HISTOGRAMS, ALL_PART_HISTOGRAMS, and DBA_PART_HISTOGRAMS
- USER_SUBPART_HISTOGRAMS, ALL_SUBPART_HISTOGRAMS, and DBA_SUBPART_HISTOGRAMS
- TAB_COLUMNS

See Also:

- "Gathering Statistics" on page 21-13
- *Oracle8i Designing and Tuning for Performance* for more information about histograms
- *Oracle8i Supplied PL/SQL Packages Reference*

Statistics for Partitioned Schema Objects

Partitioned schema objects may contain multiple sets of statistics. They can have statistics which refer to the following:

- The entire schema object (global statistics)

- An individual partition
- An individual subpartition of a composite partitioned object

Unless the query predicate narrows the query to a single partition, the optimizer uses the global statistics. Since most queries are not likely to be this restrictive, it is most important to have accurate global statistics. Intuitively, it may seem that generating global statistics from partition-level statistics should be straightforward; however, this is only true for some of the statistics. For example, it is very difficult to figure out the number of distinct values for a column from the number of distinct values found in each partition because of the possible overlap in values. Therefore, actually gathering global statistics with the DBMS_STATS package is highly recommended, rather than calculating them with the ANALYZE statement.

Note: Oracle currently does not gather global histogram statistics.

See Also: "The ANALYZE Statement" on page 21-15

Gathering Statistics

This section describes the different methods you can use to gather statistics.

The DBMS_STATS Package The PL/SQL package DBMS_STATS enables you to generate and manage statistics for cost-based optimization. You can use this package to gather, modify, view, and delete statistics. You can also use this package to store sets of statistics.

The DBMS_STATS package can gather statistics on indexes, tables, columns, and partitions, as well as statistics on all schema objects in a schema or database. It does not gather cluster statistics—you can use DBMS_STATS to gather statistics on the individual tables instead of the whole cluster.

The statistics-gathering operations can run either serially or in parallel. Whenever possible, DBMS_STATS calls a parallel query to gather statistics with the specified degree of parallelism; otherwise, it calls a serial query or the ANALYZE statement. Index statistics are not gathered in parallel.

The statistics can be computed exactly or estimated from a random sampling of rows or blocks.

For partitioned tables and indexes, DBMS_STATS can gather separate statistics for each partition as well as global statistics for the entire table or index. Similarly, for composite partitioning DBMS_STATS can gather separate statistics for

subpartitions, partitions, and the entire table or index. Depending on the SQL statement being optimized, the optimizer may choose to use either the partition (or subpartition) statistics or the global statistics.

DBMS_STATS gathers statistics only for cost-based optimization. It does not gather other statistics. For example, the table statistics gathered by DBMS_STATS include the following:

- The number of rows
- Number of blocks currently containing data
- Average row length

The table statistics gathered by DBMS_STATS do not include the following:

- The number of chained rows
- Average free space
- Number of unused data blocks

See Also:

- "Statistics Tables" on page 21-16
- "Exact and Estimated Statistics" on page 21-15
- *Oracle8i Designing and Tuning for Performance* for examples of how to gather statistics with the DBMS_STATS package

The COMPUTE STATISTICS Clause for Indexes Oracle can gather some statistics automatically while creating or rebuilding a B-tree or bitmap index. The COMPUTE STATISTICS clause of CREATE INDEX or ALTER INDEX ... REBUILD enables this gathering of statistics.

The statistics that Oracle gathers for the COMPUTE STATISTICS clause depend on whether the index is partitioned or nonpartitioned:

- For a nonpartitioned index, Oracle gathers index, table, and column statistics while creating or rebuilding the index. In a concatenated-key index, the column statistics refer only to the leading column of the key.
- For a partitioned index, Oracle does not gather any table or column statistics while creating the index or rebuilding its partitions.
  - While creating a partitioned index, Oracle gathers index statistics for each partition and for the entire index. If the index uses composite partitioning, Oracle also gathers statistics for each subpartition.

  - While rebuilding a partition or subpartition of an index, Oracle gathers index statistics only for that partition or subpartition.

To ensure correctness of the statistics Oracle always uses base tables when creating an index with the COMPUTE STATISTICS clause, even if another index is available that could be used to create the index.

> **See Also:** *Oracle8i SQL Reference* for details about the COMPUTE STATISTICS clause of the CREATE INDEX and ALTER INDEX statements

The ANALYZE Statement The ANALYZE statement can also generate statistics for cost-based optimization. Using ANALYZE for this purpose is not recommended because of various restrictions, for example:

- ANALYZE always runs serially.
- ANALYZE calculates global statistics for partitioned tables and indexes instead of gathering them directly. This can lead to inaccuracies for some statistics, such as the number of distinct values.
  - For partitioned tables and indexes, ANALYZE gathers statistics for the individual partitions and then calculates the global statistics from the partition statistics.
  - For composite partitioning, ANALYZE gathers statistics for the subpartitions and then calculates the partition statistics and global statistics from the subpartition statistics.
- ANALYZE cannot overwrite or delete some of the values of statistics that were gathered by DBMS_STATS.

ANALYZE can gather additional information that is not used by the optimizer, such as information about chained rows and the structural integrity of indexes, tables, and clusters. DBMS_STATS does not gather this information.

> **See Also:** *Oracle8i SQL Reference* for detailed information about the ANALYZE statement

Exact and Estimated Statistics The statistics gathered by the DBMS_STATS package or ANALYZE statement can be exact or estimated. (The COMPUTE STATISTICS clause for creating or rebuilding indexes always gathers exact statistics.)

To compute exact statistics, Oracle must read all of the data in the index, table, partition, or schema. Some statistics are always computed exactly, such as the

number of data blocks currently containing data in a table or the depth of an index from its root block to its leaf blocks.

To estimate statistics, Oracle selects a random sample of data. You can specify the sampling percentage and whether sampling should be based on rows or blocks.

Row sampling reads rows without regard to their physical placement on disk. This method provides the most random data for estimates, but it can result in reading more data than necessary. For example, in the worst case a row sample might select one row from each block, requiring a full scan of the table or index.

Block sampling reads a random sample of blocks and uses all of the rows in those blocks for estimates. This method reduces the amount of I/O activity for a given sample size, but it can reduce the randomness of the sample if rows are not randomly distributed on disk. Block sampling is not available for index statistics.

Managing Statistics

This section describes statistics tables and lists the views that display information about statistics stored in the data dictionary.

Statistics Tables The DBMS_STATS package enables you to store statistics in a *statistics table*. You can transfer the statistics for a column, table, index, or schema into a statistics table and subsequently restore those statistics to the data dictionary. The optimizer does not use statistics that are stored in a statistics table.

Statistics tables enable you to experiment with different sets of statistics. For example, you can back up a set of statistics before you delete them, modify them, or generate new statistics. You can then compare the performance of SQL statements optimized with different sets of statistics, and if the statistics stored in a table give the best performance, you can restore them to the data dictionary.

A statistics table can keep multiple distinct sets of statistics, or you can create multiple statistics tables to store distinct sets of statistics separately.

Viewing Statistics You can use the DBMS_STATS package to view the statistics stored in the data dictionary or in a statistics table.

You can also query these data dictionary views for statistics in the data dictionary:

- USER_TABLES, ALL_TABLES, and DBA_TABLES
- USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS
- USER_INDEXES, ALL_INDEXES, and DBA_INDEXES
- USER_CLUSTERS and DBA_CLUSTERS

- USER_TAB_PARTITIONS, ALL_TAB_PARTITIONS, and DBA_TAB_PARTITIONS
- USER_TAB_SUBPARTITIONS, ALL_TAB_SUBPARTITIONS, and DBA_TAB_SUBPARTITIONS
- USER_IND_PARTITIONS, ALL_IND_PARTITIONS, and DBA_IND_PARTITIONS
- USER_IND_SUBPARTITIONS, ALL_IND_SUBPARTITIONS, and DBA_IND_SUBPARTITIONS
- USER_PART_COL_STATISTICS, ALL_PART_COL_STATISTICS, and DBA_PART_COL_STATISTICS
- USER_SUBPART_COL_STATISTICS, ALL_SUBPART_COL_STATISTICS, and DBA_SUBPART_COL_STATISTICS

> **See Also:** *Oracle8i Reference*. for information on the statistics in these views

When to Use the Cost-Based Approach

In general, you should use the cost-based approach for all new applications; the rule-based approach is provided for applications that were written before cost-based optimization was available. Cost-based optimization can be used for both relational data and object types.

The following features can only use cost-based optimization:

- Partitioned tables
- Partition views
- Index-organized tables
- Reverse key indexes
- Bitmap indexes
- Function-based indexes
- SAMPLE clause in a SELECT statement
- Parallel query and parallel DML
- Star transformation
- Star join

- Extensible optimization

  See Also: *Oracle8i Designing and Tuning for Performance* for more information on when to use the cost-based approach

Extensible Optimization

Extensible optimization allows the authors of user-defined functions and domain indexes to control the three main components that cost-based optimization uses to select an execution plan: statistics, selectivity, and cost evaluation.

Extensible optimization allows you to:

- Associate cost function and default costs with domain indexes, indextypes, packages, and stand-alone functions
- Associate selectivity function and default selectivity with methods of object types, package functions, and stand-alone functions
- Associate statistics collection functions with domain indexes and columns of tables
- Order predicates with functions based on cost
- Select a user-defined access method (domain index) for a table based on access cost
- Use the ANALYZE statement to invoke user-defined statistics collection and deletion functions
- Use new data dictionary views to include information about the statistics collection, cost, or selectivity functions associated with columns, domain indexes, indextypes, or functions
- Add a hint to preserve the order of evaluation for function predicates

  See Also: *Oracle8i Data Cartridge Developer's Guide* for details about extensible optimization

User-Defined Statistics

You can define *statistics collection functions* for domain indexes, individual columns of a table, and user-defined datatypes.

Whenever a domain index is analyzed to gather statistics, Oracle calls the associated statistics collection function. Whenever a column of a table is analyzed, Oracle collects the standard statistics for that column and calls any associated statistics collection function. If a statistics collection function exists for a datatype, then Oracle calls it for each column that has that datatype in the table being analyzed.

User-Defined Selectivity

The selectivity of a predicate in a SQL statement is used to estimate the cost of a particular access method; it is also used to determine the optimal join order. The optimizer cannot compute an accurate selectivity for predicates that contain user-defined operators because it does not have any information about these operators.

You can define *selectivity functions* for predicates containing user-defined operators, stand-alone functions, package functions, or type methods. The optimizer calls the user-defined selectivity function whenever it encounters a predicate that contains the operator, function, or method in one of the following relations with a constant: <, <=, =, >=, >, or LIKE.

User-Defined Costs

The optimizer cannot compute an accurate estimate of the cost of a domain index because it does not know the internal storage structure of the index. Also, the optimizer may underestimate the cost of a user-defined function with any of the following characteristics:

- Invokes PL/SQL
- Uses recursive SQL
- Accesses a BFILE
- Uses a great deal of CPU

You can define costs for the following:

- Domain indexes
- User-defined stand-alone functions
- Package functions

- Type methods

These user-defined costs can be in the form of default costs that the optimizer simply looks up or they can be full-fledged cost functions that the optimizer calls to compute the cost.

Rule-Based Optimization

Using the rule-based approach, the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths. You can use rule-based optimization to access both relational data and object types.

Oracle's ranking of the access paths is heuristic. If there is more than one way to execute a SQL statement, the rule-based approach always uses the operation with the lower rank. Usually, operations of lower rank execute faster than those associated with constructs of higher rank.

Note: Rule-based optimization is not available for some advanced features of Oracle8*i*.

See Also:

- *Oracle8i Designing and Tuning for Performance*
- "When to Use the Cost-Based Approach" on page 21-17 for a list of features for which rule-based optimization is not available

Part VII

Parallel SQL and Direct-Load INSERT

Part VII describes parallel execution of SQL statements and the direct-load INSERT feature. It contains the following chapters:

22

Direct-Load INSERT

This chapter describes the Oracle direct-load INSERT feature for serial or parallel inserts. It also describes the NOLOGGING feature that is available for direct-load INSERT and some DDL statements. This chapter's topics include:

- Introduction to Direct-Load INSERT
- Varieties of Direct-Load INSERT Statements
  - Serial and Parallel INSERT
  - Logging Mode
- Additional Considerations for Direct-Load INSERT
- Restrictions on Direct-Load INSERT

Note: The parallel direct-load INSERT feature described in this chapter is available only if you have purchased the Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for more information.

See Also:

- Chapter 23, "Parallel Execution of SQL Statements" for more information about parallel execution INSERT issues
- *Oracle8i Designing and Tuning for Performance* for information on how to tune parallel direct-load INSERT

Introduction to Direct-Load INSERT

Direct-load INSERT enhances performance during insert operations by formatting and writing data directly into Oracle datafiles, without using the buffer cache. This functionality is similar to that of the Direct Loader utility (SQL*Loader).

Direct-load INSERT appends the inserted data after existing data in a table; free space within the existing data is not reused. Data can be inserted into partitioned or nonpartitioned tables, either in parallel or serially.

Several options of direct-load INSERT exist with respect to parallelism, table partitioning, and logging.

See Also:

- "Varieties of Direct-Load INSERT Statements" on page 22-3
- Chapter 23, "Parallel Execution of SQL Statements" for more information about the parallelism and partitioning options of direct-load INSERT

Advantages of Direct-Load INSERT

A major benefit of direct-load INSERT is that you can load data without logging redo or undo entries, which improves the insert performance significantly. Both serial and parallel direct-load INSERT have this performance advantage over conventional path INSERT.

With the conventional path INSERT, in contrast, free space in the object is reused and referential integrity can be maintained. The conventional path for insertions cannot be parallelized.

Comparison with CREATE TABLE ... AS SELECT

With direct-load INSERT, you can insert data into existing tables instead of having to create new tables. Direct-load INSERT updates the indexes of the table, but CREATE TABLE ... AS SELECT only creates a new table which does not have any indexes.

See Also: "CREATE TABLE ... AS SELECT in Parallel" on page 23-32

Advantage over Parallel Direct Load (SQL*Loader)

With a parallel INSERT, atomicity of the transaction is ensured. Atomicity cannot be guaranteed if multiple parallel loads are used. Also, parallel load could leave some table indexes in an UNUSABLE state at the end of the load if errors occurred while updating the indexes. In comparison, parallel INSERT atomically updates the table and indexes (that is, it rolls back the statement if errors occur while updating the index).

> **See Also:** *Oracle8i Utilities* for information about parallel load

INSERT ... SELECT Statements

Direct-load INSERT (serial or parallel) can only support the INSERT ... SELECT syntax of an INSERT statement, not the INSERT... *values* syntax. The parallelism for INSERT ... SELECT is determined from either parallel hints or parallel table definition clauses.

> **See Also:** *Oracle8i SQL Reference* for information about the syntax of INSERT ... SELECT statements

Varieties of Direct-Load INSERT Statements

Direct-load INSERT can be performed in the following ways:

- Serially or in parallel
- Into partitioned or nonpartitioned tables
- With or without logging of redo data

Serial and Parallel INSERT

Direct-load INSERT can be done on partitioned or nonpartitioned tables, and it can be done either serially or in parallel.

- **Serial direct-load INSERT into a nonpartitioned or partitioned table.** Data is inserted beyond the current high water mark of the table segment or 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 others.
- **Parallel direct-load INSERT into a nonpartitioned table.** Each parallel execution server allocates a new temporary segment and inserts data into the temporary segment. When a commit executes, the parallel execution

coordinator merges the new temporary segments into the primary table segment.

- **Parallel direct-load INSERT into a partitioned table.** Each parallel execution server is assigned one or more partitions, with no more than one process working on any given partition. The parallel execution server inserts data beyond the current high water mark of the partition segments assigned to it. When a commit executes, the high water mark of each partition segment is updated by the parallel execution coordinator to the new value, making the data visible to others.

In all the cases, the bumping of the high water mark or merging of the temporary segment is delayed until commit is issued, because this action immediately makes the data visible to other processes. That is, it commits the insert operation.

See Also: "Process Architecture for Parallel Execution" on page 23-5

Specifying Serial or Parallel Direct-Load INSERT

The APPEND hint is required for using serial direct-load INSERT. Parallel direct-load INSERT requires either a PARALLEL hint in the statement or a PARALLEL clause in the table definition; the APPEND hint is optional. Parallel direct-load INSERT also requires parallel DML to be enabled with the ALTER SESSION ENABLE/FORCE PARALLEL DML statement.

Table 22–1 summarizes these requirements and compares direct-load INSERT with conventional INSERT.

Table 22–1 Summary of Serial and Parallel INSERT ... SELECT Statements

Insert Type	Serial	Parallel
Direct-load INSERT	Yes: requires APPEND hint in SQL statement	Yes: requires ■ ALTER SESSION ENABLE/FORCE PARALLEL DML ■ table PARALLEL attribute or statement PARALLEL hint ■ an APPEND hint is optional
Conventional INSERT	Yes (default)	No

Examples of Serial and Parallel Direct-Load INSERT

You can specify serial direct-load INSERT with the APPEND hint, for example:

```
INSERT /*+ APPEND */ INTO emp
   SELECT * FROM t_emp;
COMMIT;
```

You can specify parallel direct-load INSERT by setting the PARALLEL attribute of the table into which rows are inserted, for example:

```
ALTER TABLE emp PARALLEL (10);
ALTER SESSION ENABLE PARALLEL DML;
INSERT INTO emp
   SELECT * FROM t_emp;
COMMIT;
```

You can also specify parallelism for the SELECT operation by setting the PARALLEL attribute of the table from which rows are selected:

```
ALTER TABLE emp PARALLEL (10);
ALTER TABLE t_emp PARALLEL (10);
ALTER SESSION ENABLE PARALLEL DML;
INSERT INTO emp
   SELECT * FROM t_emp;
COMMIT;
```

The PARALLEL hint for an INSERT or SELECT operation takes precedence over a table's PARALLEL attribute. For example, the degree of parallelism in the following INSERT ... SELECT statement is 12 regardless of whether the PARALLEL attributes are set for the EMP and T_EMP tables:

```
ALTER SESSION ENABLE PARALLEL DML;
INSERT /*+ PARALLEL(emp,12) */ INTO emp
   SELECT /*+ PARALLEL(t_emp,12) */ * FROM t_emp;
COMMIT;
```

See Also: "Rules for Parallelizing INSERT ... SELECT" on page 23-23

Logging Mode

Direct-load INSERT operations can be done with or without logging of redo information. You can specify no-logging mode for the table, partition, or index into which data will be inserted by using an ALTER TABLE, ALTER INDEX, or ALTER TABLESPACE statement.

- **Direct-load INSERT with logging.** This mode can do full redo logging for instance and media recovery. Logging is the default mode. The database must be in ARCHIVELOG mode for online redo logs to be archived to tape. Otherwise, instance crashes are recoverable, but disk failures are not recoverable.
- **Direct-load INSERT with no-logging.** In this mode, data is inserted without redo or undo logging. (Some minimal logging is still done for marking new extents invalid, and dictionary changes are always fully logged.) When applied during media recovery, the *extent invalidation* records mark a range of blocks as logically corrupt, since the redo data is not logged.

The no-logging mode improves performance because it generates much less log data. The user is responsible for backing up the data after a no-logging insert operation in order to be able to perform media recovery.

There is no interaction between no-logging mode and discrete transactions, which always generate redo information. Discrete transactions can be issued against tables that use the no-logging attribute.

Note: Logging / no-logging mode is not a permanent attribute of the table, partition, or index. After the database object inserted into has been populated with data and backed up, you can set its status to logging mode so that subsequent changes will be logged.

Table 22–2 compares the LOGGING and NOLOGGING modes for direct-load and conventional INSERT.

Table 22–2 Summary of LOGGING and NOLOGGING clauses

Insert Type	LOGGING	NOLOGGING
Direct-load INSERT	Yes: recoverability requires ARCHIVELOG database mode	Yes: requires NOLOGGING attribute for tablespace, table, partition, or index
Conventional INSERT	Yes (default): recoverability requires ARCHIVELOG database mode	No

See Also: "Discrete Transaction Management" on page 16-9

Examples of No-Logging Mode

You can specify no-logging mode for direct-load INSERT by setting the NOLOGGING attribute of the table into which rows are inserted, for example:

```
ALTER TABLE emp NOLOGGING;
ALTER SESSION ENABLE PARALLEL DML;
INSERT /*+ PARALLEL(emp,12) */ INTO emp
   SELECT /*+ PARALLEL(t_emp,12) */ * FROM t_emp;
COMMIT;
```

You can also set the NOLOGGING attribute for a partition, tablespace, or index; for example:

```
ALTER TABLE emp MODIFY PARTITION emp_lmnop NOLOGGING;

ALTER TABLESPACE personnel NOLOGGING;

ALTER INDEX emp_ix NOLOGGING;

ALTER INDEX emp_ix MODIFY PARTITION eix_lmnop NOLOGGING;
```

SQL Statements That Can Use No-Logging Mode

Although you can set the NOLOGGING attribute for a table, partition, index, or tablespace, no-logging mode does not apply to every operation performed on the schema object for which you set the NOLOGGING attribute. Only the following operations can make use of no-logging mode:

- direct load (SQL*Loader)
- direct-load INSERT
- CREATE TABLE ... AS SELECT
- CREATE INDEX
- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER INDEX ... SPLIT PARTITION
- ALTER INDEX ... REBUILD
- ALTER INDEX ... REBUILD PARTITION
- INSERT, UPDATE, and DELETE on LOBs in NOCACHE NOLOGGING mode stored out of line

All of these SQL statements can be parallelized. They can execute in logging or no-logging mode for both serial and parallel execution.

Other SQL statements are unaffected by the NOLOGGING attribute of the schema object. For example, the following SQL statements are unaffected by NOLOGGING mode: UPDATE and DELETE (except on some LOBs, as noted above), conventional path INSERT, and various DDL statements not listed above.

See Also: Chapter 23, "Parallel Execution of SQL Statements"

Default Logging Mode

If the LOGGING or NOLOGGING clause is not specified, the logging attribute of the table, partition, or index defaults to the logging attribute of the tablespace in which it resides.

For LOBs, if the LOGGING or NOLOGGING clause is omitted, then:

- If CACHE is specified, then LOGGING is used because LOBs cannot have CACHE NOLOGGING
- Otherwise, the default is obtained from the tablespace in which the LOB value resides

Additional Considerations for Direct-Load INSERT

This section describes index maintenance, space allocation, and data locks for direct-load INSERT.

Index Maintenance

For direct-load INSERT on nonpartitioned tables or partitioned tables that have local or global indexes, index maintenance is done at the end of the INSERT operation. This index maintenance is performed by the parallel execution servers for parallel direct-load INSERT or by the single process for serial direct-load INSERT on partitioned or nonpartitioned tables.

If your direct-load INSERT modifies most of the data in a table, you can avoid the performance impact of index maintenance by dropping the index before the INSERT and then rebuilding it afterwards.

Space Considerations

Direct-load INSERT requires more space than conventional path INSERT, because direct-load INSERT ignores existing space in the free lists of the segment. For parallel direct-load INSERT into nonpartitioned tables, free blocks above the high water mark of the table segment are also ignored. Additional space requirements must be considered before using direct-load INSERT.

Parallel direct-load INSERT into a nonpartitioned table creates temporary segments—one segment for each degree of parallelism. For example, if you use parallel INSERT into a nonpartitioned table with the degree of parallelism set to four, then four temporary segments are created.

Each parallel execution server first inserts its data into a temporary segment, and finally the data in all of the temporary segments is appended to the table. (This is the same mechanism as CREATE TABLE ... AS SELECT.)

For parallel INSERT into a partitioned table, no temporary segments are created. Each parallel execution server simply inserts its data into a partition above the high water mark.

When you are doing a parallel INSERT for a nonpartitioned table that is not locally managed and is not in automatic mode, modifying the values of the following parameters allows you to provide sufficient storage for temporary segments without wasting space on segments that are larger than you need:

NEXT	the size, in bytes, of the object's next extent to be allocated to the object
PCTINCREASE	the percentage by which the third and subsequent extents grow over the preceding extent
MINIMUM EXTENT	controls free space fragmentation in the tablespace by ensuring that every used or free extent size in a tablespace is at least as large as, and is a multiple of, the value you specify

Choose values for these parameters such that:

- The size of each extent is not too small (no less than 1 MB). This affects the total number of extents in the object.
- The size of each extent is not so large that the parallel INSERT results in wasting space on segments that are larger than necessary.

You can change the values of the NEXT and PCTINCREASE parameters with the STORAGE clause of the ALTER TABLE statement. You can change the value of the MINIMUM EXTENT parameter with the ALTER TABLESPACE statement. After performing the parallel DML statement, you can change the values of the NEXT,

PCTINCREASE, and MINIMUM EXTENT parameters back to settings appropriate for non-parallel operations.

Locking Considerations

In direct-load INSERT, exclusive locks are obtained on the table (or on all the partitions of a partitioned table) precluding any concurrent insert, update, or delete on the table. Concurrent queries, however, are supported and will see only the data in the table before the INSERT began. These locks also prevent any concurrent index creation or rebuild operations. This must be taken into account before using direct-load INSERT because it affects table concurrency.

See Also: "Lock and Enqueue Resources for Parallel DML" on page 23-43

Restrictions on Direct-Load INSERT

The restrictions on direct-load INSERT are the same as those imposed on direct-path parallel loading with SQL*Loader, because they use the same underlying mechanism. In addition, the general parallel DML restrictions also apply to direct-load INSERT.

Serial and parallel direct-load INSERT have the following restrictions:

- A transaction can contain multiple direct-load INSERT statements (or both direct-load INSERT statements and parallel UPDATE or DELETE statements), but after one of these statements modifies a table, no other SQL statement in the transaction can access the same table.
  - Queries that access the same table are allowed before the direct-load INSERT statement, but not after.
  - Any serial or parallel statements attempting to access a table that has already been modified by a direct-load INSERT (or parallel DML) within the same transaction are rejected with an error message.
- If the initialization parameter ROW_LOCKING = INTENT, then inserts cannot be performed by the direct-load path.
- Direct-load INSERT does not support referential integrity.
- Triggers are not supported for direct-load INSERT operations.
- Replication functionality is not supported for direct-load INSERT.

- Direct-load INSERT cannot occur on tables with object columns or LOB columns, or on index-organized tables.
- A transaction involved in a direct-load INSERT operation cannot be or become a distributed transaction.
- Clustered tables are not supported.

Violations of the restrictions will cause the statement to execute serially, using the conventional insert path, without warnings or error messages. An exception is the restriction on statements accessing the same table more than once in a transaction, which can cause error messages.

For example, if triggers or referential integrity are present on the table, then the APPEND hint will be ignored when you try to use direct-load INSERT (serial or parallel), as well as the PARALLEL hint or clause, if any.

See Also: "Restrictions on Parallel DML" on page 23-44

23

Parallel Execution of SQL Statements

This chapter describes the parallel execution of SQL statements. The topics in this chapter include:

Note: The parallel execution features described in this chapter are available only if you have purchased the Oracle8*i* Enterprise Edition. See *Getting to Know Oracle8i* for information about Oracle8*i* Enterprise Edition.

Also, parallel execution is not the same as the Oracle Parallel Server. You do not need the Parallel Server Option to perform parallel execution of SQL statements; however, some aspects of parallel execution apply only to the Oracle Parallel Server.

Introduction to Parallel Execution of SQL Statements

When Oracle is not parallelizing the execution of SQL statements, each SQL statement is executed sequentially by a single process. With parallel execution, however, 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.

Parallel execution can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. 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.

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.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for specific information on tuning your parameter files and database to take full advantage of parallel execution

Operations That Can Be Parallelized

The Oracle server can use parallel execution for any of these operations:

- Table scan
- Nested loop join
- Sort merge join
- Hash join
- "Not in"
- Group by
- Select distinct
- Union and union all
- Aggregation
- PL/SQL functions called from SQL

- Order by
- Create table as select
- Create index
- Rebuild index
- Rebuild index partition
- Move partition
- Split partition
- Update
- Delete
- Insert ... select
- Enable constraint (the table scan is parallelized)
- Star transformation
- Cube
- Rollup

How Oracle Parallelizes Operations

A SELECT statement contains a query only. A DML or DDL statement usually contains a query portion and a DML or DDL portion, each of which can be parallelized.

> **Note:** Although data manipulation language (DML) includes queries, in this chapter "DML" refers only to INSERTs, UPDATEs, and DELETEs.

Oracle primarily parallelizes SQL statements in the following ways:

1. Parallelize by block ranges for scan operations (SELECTs and subqueries in DML and DDL statements).
2. Parallelize by partitions for DDL and DML operations on partitioned tables and indexes.
3. Parallelize by parallel execution servers for inserts into nonpartitioned tables only.

Parallelizing by Block Range

Oracle parallelizes a query dynamically at execution time. *Dynamic parallelism* divides the table or index into ranges of database blocks *(rowid range)* and executes the operation in parallel on different ranges. If the distribution or location of data changes, Oracle automatically adapts to optimize the parallelization for each execution of the query portion of a SQL statement.

Parallel scans by block range break the table or index into pieces delimited by high and low rowid values. The table or index can be nonpartitioned or partitioned.

For partitioned tables and indexes, no rowid range can span a partition although one partition can contain multiple rowid ranges. Oracle sends the partition numbers with the rowid ranges to avoid partition map lookup. Compile and run-time predicates on partitioning columns restrict the rowid ranges to relevant partitions, eliminating unnecessary partition scans (*partition pruning*).

This means that a parallel query which accesses a partitioned table by a table scan performs the same or less overall work as the same query on a nonpartitioned table. The query on the partitioned table executes with equivalent parallelism, although the total number of disks accessed might be reduced by the partition pruning.

Oracle can parallelize the following operations on tables and indexes by block range (rowid range):

- Queries using table scans (including queries in DML and DDL statements)
- Move partition
- Split partition
- Rebuild index partition
- Create index (nonpartitioned index)
- Create table ... as select (nonpartitioned table)

Parallelizing by Partition

Partitions are a logical static division of tables and indexes which can be used to break some long-running operations into smaller operations executed in parallel on individual partitions. The granule of parallelism is a partition; there is no parallelism within a partition except for:

- Queries, which can be parallelized by block range as described above
- Composite partitioning, in which the granule of parallelism is a subpartition

Operations on partitioned tables and indexes are performed in parallel by assigning different parallel execution servers to different partitions of the table or index. Compile and run-time predicates restrict the partitions when the operation references partitioning columns. The operation executes serially when compile or run-time predicates restrict the operation to a single partition.

The parallel operation may use fewer parallel execution servers than the number of accessed partitions (because of resource limits, hints, or table attributes), but each partition is accessed by a single parallel execution server. A parallel execution server, however, can access multiple partitions.

Operations on partitioned tables and indexes are performed in parallel only when more than one partition is accessed.

Oracle can parallelize the following operations on partitioned tables and indexes by partition:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- UPDATE
- DELETE
- INSERT ... SELECT
- ALTER INDEX ... REBUILD
- Queries using a range scan on a partitioned index

See Also: "Composite Partitioning" on page 11-16

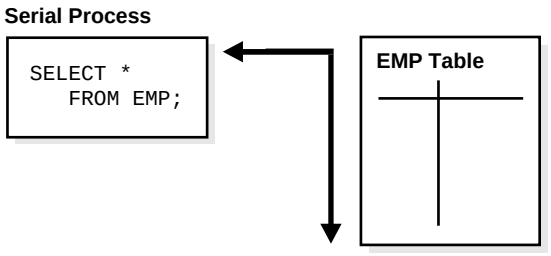
Parallelizing by Parallel Execution Servers

For nonpartitioned tables only, Oracle parallelizes insert operations by dividing the work among parallel execution servers. Because new rows do not have rowids, the rows are distributed among the parallel execution servers to insert them into the free space.

Process Architecture for Parallel Execution

When parallel execution is not 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 23–1.

Figure 23–1 Serial Full Table Scan



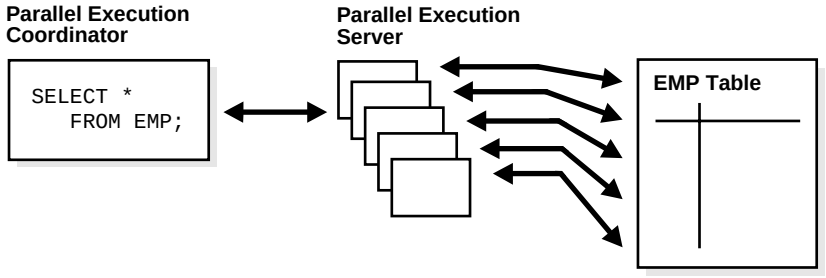
Parallel execution performs these operations in parallel using multiple *parallel processes*. One process, known as the *parallel execution coordinator*, dispatches the execution of a statement to several *parallel execution servers* and coordinates the results from all of the server processes to send the results back to the user.

Note: *Parallel execution server* does not mean a process of an Oracle Parallel Server, but instead means a process that performs an operation in parallel with other processes. In an Oracle Parallel Server, the parallel execution servers may be spread across multiple instances. Parallel execution servers are also sometimes called *slave processes*.

When an operation is divided into pieces for parallel execution in a massively parallel processing (MPP) configuration, Oracle assigns a particular piece of the operation to a parallel execution server by taking into account the *affinity* of the process for the piece of the table or index to be used for the operation. The physical layout of partitioned tables and indexes impacts on the affinity used to assign work for parallel execution servers.

Figure 23–2 illustrates several parallel execution servers simultaneously performing a partial scan of the EMP table, which is divided by block range dynamically *(dynamic partitioning)*. The parallel execution servers send results back to the parallel execution coordinator, which assembles the pieces into the desired full table scan.

Figure 23–2 Parallel Full Table Scan



The parallel execution coordinator breaks down execution functions into parallel pieces and then integrates the partial results produced by the parallel execution servers. 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:

- "Affinity" on page 23-49
- "How Oracle Determines the Degree of Parallelism for Operations" on page 23-16

The Parallel Execution Server Pool

When an instance starts up, Oracle creates a pool of parallel execution servers which are available for any parallel operation. The initialization parameter PARALLEL_MIN_SERVERS specifies the number of parallel execution servers that Oracle creates at instance startup.

When executing a parallel operation, the parallel execution coordinator obtains parallel execution servers from the pool and assigns them to the operation. If necessary, Oracle can create additional parallel execution servers for the operation. These parallel execution servers remain with the operation throughout job execution, then become available for other operations. After the statement has been processed completely, the parallel execution servers return to the pool.

> **Note:** The parallel execution coordinator and the parallel execution servers can only service one statement at a time. A parallel execution coordinator cannot coordinate, for example, a parallel query and a parallel DML statement at the same time.

When a user issues a SQL statement, the optimizer decides whether to execute the operations in parallel and determines the degree of parallelism for each operation. You can specify the number of parallel execution servers required for an operation in various ways.

If the optimizer targets the statement for parallel processing, the following sequence of events takes place:

1. The SQL statement's foreground process becomes a parallel execution coordinator.
2. The parallel execution coordinator obtains as many parallel execution servers as needed (determined by the degree of parallelism) from the server pool or creates new parallel execution servers as needed.
3. Oracle executes the statement as a sequence of operations. Each operation is performed in parallel, if possible.
4. When statement processing is completed, the coordinator returns any resulting data to the user process that issued the statement and returns the parallel execution servers to the server pool.

The parallel execution coordinator calls upon the parallel execution servers during the execution of the SQL statement, not during the parsing of the statement. Therefore, when parallel execution is used with the multi-threaded server, the server process that processes the EXECUTE call of a user's statement becomes the parallel execution coordinator for the statement.

See Also: "Setting the Degree of Parallelism" on page 23-15

Variations in the Number of Parallel Execution Servers

If the number of parallel operations processed concurrently by an instance changes significantly, Oracle automatically changes the number of parallel execution servers in the pool.

If the number of parallel operations increases, Oracle creates additional parallel execution servers to handle incoming requests. However, Oracle never creates more

parallel execution servers for an instance than what is specified by the initialization parameter PARALLEL_MAX_SERVERS.

If the number of parallel operations decreases, Oracle terminates any parallel execution servers that have been idle for a threshold period of time. Oracle does not reduce the size of the pool below the value of PARALLEL_MIN_SERVERS no matter how long the parallel execution servers have been idle.

Processing Without Enough Parallel Execution Servers

Oracle can process a parallel operation with fewer than the requested number of processes.

If all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started, the parallel execution coordinator switches to serial processing.

See Also:

- "Minimum Number of Parallel Execution Servers" on page 23-18 for information about specifying a minimum with the initialization parameter PARALLEL_MIN_PERCENT
- *Oracle8i Designing and Tuning for Performance* for information about monitoring an instance's pool of parallel execution servers and determining the appropriate values of the initialization parameters

How Parallel Execution Servers Communicate

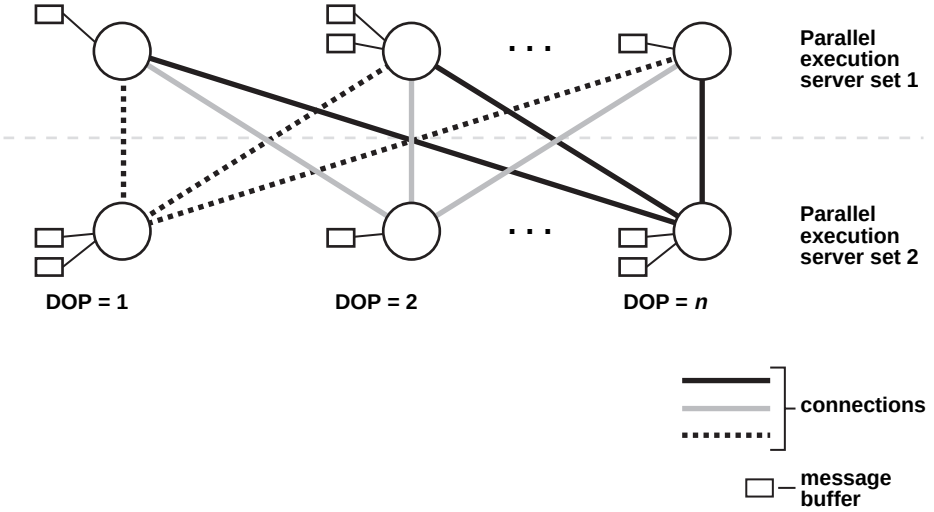
To execute a query in parallel, Oracle generally creates a producer queue server and a consumer server. The producer queue server retrieves rows from tables and the consumer server performs operations such as join, sort, DML, and DDL on these rows. Each server in the producer execution process set has a connection to each server in the consumer set. This means that the number of virtual connections between parallel execution servers increases as the square of the degree of parallelism.

Each communication channel has at least 1, and sometimes up to 4 memory buffers. Multiple memory buffers facilitate asynchronous communication among the parallel execution servers.

A single-instance environment uses at most 3 buffers per communication channel. An Oracle Parallel Server environment uses at most 4 buffers per channel.

Figure 23–3 illustrates message buffers and how producer parallel execution servers connect to consumer parallel execution servers.

Figure 23–3 Parallel Execution Server Connections and Buffers



When a connection is between two processes on the same instance, the servers communicate by passing the buffers back and forth. When the connection is between processes in different instances, the messages are sent using external high-speed network protocols. In Figure 23–3, the degree of parallelism is equal to the number of parallel execution servers, which in this case is "n". Figure 23–3 does not show the parallel execution coordinator: each parallel execution server actually has an additional connection to the parallel execution coordinator.

Parallelizing 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. For example, parallelize a full table scan by block range or

parallelize an index range scan by partition. 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 is the *degree of parallelism*.

See Also:

- Chapter 21, "The Optimizer"
- "Setting the Degree of Parallelism" on page 23-15
- "Parallelization Rules for SQL Statements" on page 23-20

Dividing Work Among Parallel Execution Servers

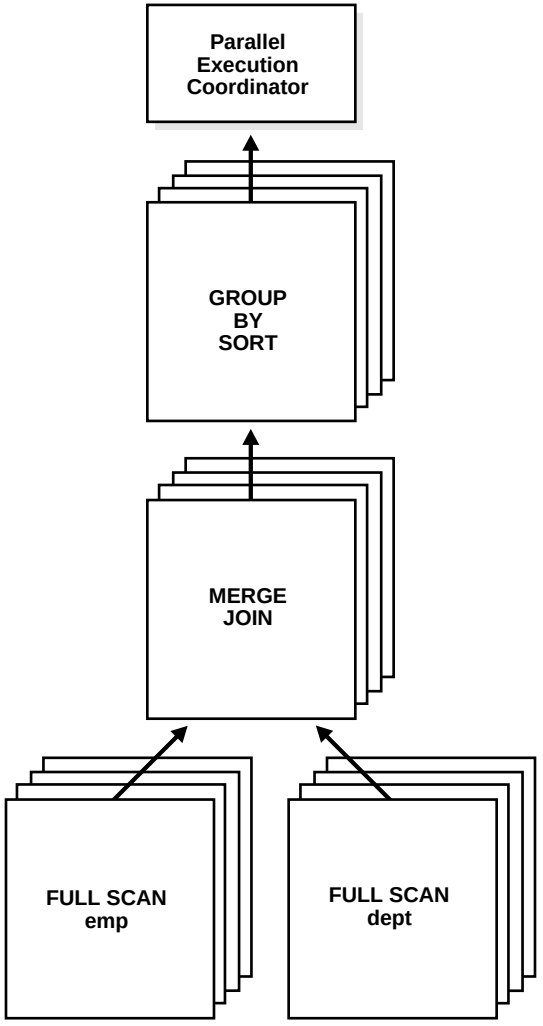
The parallel execution coordinator examines the redistribution requirements of each operation. An operation's *redistribution requirement* is the way in which the rows operated on by the operation must be divided, or redistributed, among the parallel execution servers.

After determining the redistribution requirement for each operation in the execution plan, the optimizer determines the order in which the operations in the execution plan must be performed. With this information, the optimizer determines the data flow of the statement.

Figure 23–4 illustrates the data flow of the following query:

```
SELECT dname, MAX(sal), AVG(sal)
  FROM emp, dept
    WHERE emp.deptno = dept.deptno
    GROUP BY dname;
```

Figure 23–4 Data Flow Diagram for a Join of the EMP and DEPT Tables



Parallelism Between Operations

Operations that require the output of other operations are known as *parent* operations. In Figure 23–4 the GROUP BY SORT operation is the parent of the MERGE JOIN operation because GROUP BY SORT requires the MERGE JOIN output.

Parent operations can begin consuming rows as soon as the child operations have produced rows. In the previous example, while the parallel execution servers are producing rows in the FULL SCAN DEPT operation, another set of parallel execution servers can begin to perform the MERGE JOIN operation to consume the rows.

Each of the two operations performed concurrently is given its own set of parallel execution servers. Therefore, both query operations and the data flow tree itself have parallelism. The parallelism of an individual operation is called *intra-operation* parallelism and the parallelism between operations in a data flow tree is called *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.

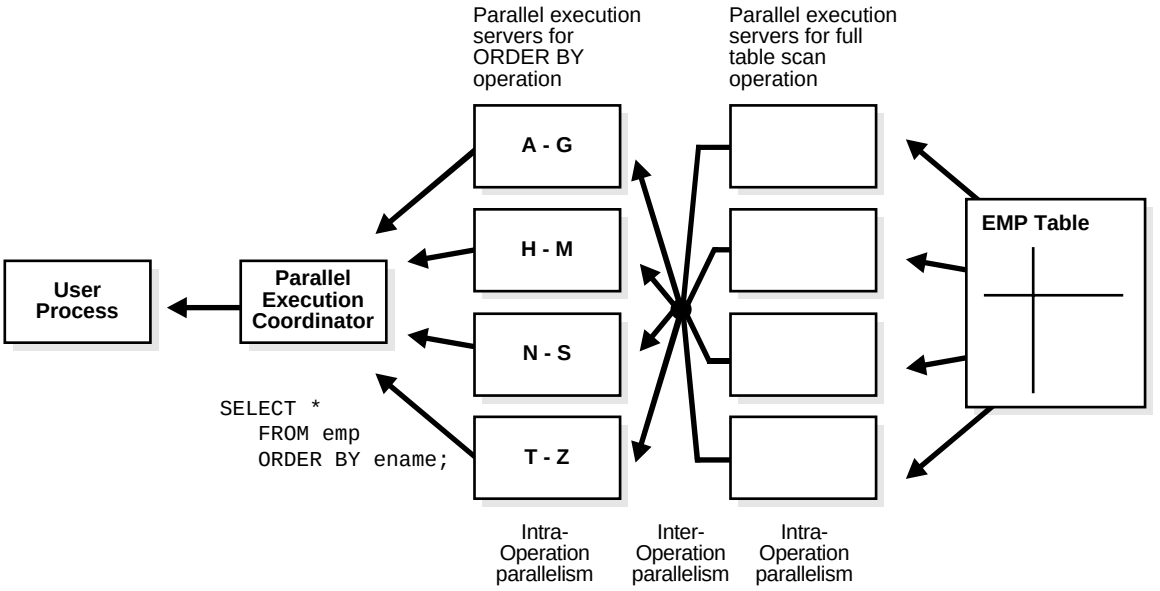
To illustrate intra-operation parallelism and inter-operator parallelism, consider the following statement:

```
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.

Figure 23–5 illustrates the parallel execution of our example query.

Figure 23–5 Inter-Operation Parallelism and Dynamic Partitioning



As you can see from Figure 23–5, 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.

Setting the 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*.

The degree of parallelism is specified in the following ways:

- At the statement level
  - With hints
  - With the PARALLEL clause
- At the table level in the table's definition
- At the index level in the index's definition
- By default based on the number of CPUs

The following example shows a statement that sets the degree of parallelism to 4 on a table:

```
ALTER TABLE emp PARALLEL 4;
```

This next example sets the degree of parallelism on an index:

```
ALTER INDEX iemp PARALLEL;
```

This last example sets a hint to 4 on a query:

```
SELECT /*+ PARALLEL(emp,4) */ COUNT(*) FROM emp ;
```

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 operations can be performed simultaneously.

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 deal with resource utilization in conjunction with parallel execution, including:

- The adaptive multi-user algorithm, which reduces the degree of parallelism as the load on the system increases. You can turn this option on with the PARALLEL_ADAPTIVE_MULTI_USER parameter of the ALTER SYSTEM statement or in your initialization 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:

- *Oracle8i Reference* and *Oracle8i Designing and Tuning for Performance* for information about the syntax of the SELECT and ALTER statements
- "User Resource Limits and Profiles" on page 26-17
- Chapter 9, "Database Resource Management"
- *Oracle8i SQL Reference* for the syntax of the ALTER SYSTEM SQL statement

How Oracle Determines the Degree of Parallelism for Operations

The parallel execution coordinator determines the degree of parallelism by considering several specifications. The coordinator:

1. Checks for hints or a PARALLEL clause specified in the SQL statement itself.
2. Looks at the table's or index's definition.
3. Checks for the default degree of parallelism.

After a degree of parallelism is found in one of these specifications, it becomes the degree of parallelism for the operation.

Hints, PARALLEL clauses, table or index definitions, and default values only determine the number of parallel execution servers that the coordinator *requests* for a given operation. The actual number of parallel execution servers uses depends

upon how many processes are available in the parallel execution server pool and whether inter-operation parallelism is possible.

See Also:

- "Default Degree of Parallelism" on page 23-17
- "Parallelization Rules for SQL Statements" on page 23-20
- "The Parallel Execution Server Pool" on page 23-7
- "Parallelism Between Operations" on page 23-13

Hints

You can specify hints in a SQL statement to set the degree of parallelism for a table or index and the caching behavior of the operation.

- The PARALLEL hint is used only for operations on tables. You can use it to parallelize queries and DML statements (INSERT, UPDATE, and DELETE).
- The PARALLEL_INDEX hint parallelizes an index range scan of a partitioned index. (In an index operation, the PARALLEL hint is not valid and is ignored.)

See Also: *Oracle8i Designing and Tuning for Performance* for a general discussion on using hints in SQL statements and the specific syntax for the PARALLEL, NOPARALLEL, PARALLEL_INDEX, CACHE, and NOCACHE hints

Table and Index Definitions

You can specify the degree of parallelism within a table or index definition. Use one of the following SQL statements to set the degree of parallelism for a table or index: CREATE TABLE, ALTER TABLE, CREATE INDEX, or ALTER INDEX.

See Also: *Oracle8i SQL Reference* for the complete syntax of SQL statements

Default Degree of Parallelism

The default degree of parallelism is used when you ask to parallelize an operation but you do not specify a degree of parallelism in a hint or within the definition of a table or index. The default degree of parallelism is appropriate for most applications.

The default degree of parallelism for a SQL statement is determined by the following factors:

- The number of CPUs for all Oracle Parallel Server instances in the system, and the value of the parameter PARALLEL_THREADS_PER_CPU
- For parallelizing by partition, the number of partitions that will be accessed, based on partition pruning
- For parallel DML operations with global index maintenance, the minimum number of transaction free lists among all the global indexes to be updated. The minimum number of transaction free lists for a partitioned global index is the minimum number across all index partitions. This is a requirement to prevent self-deadlock.

> **Note:** Oracle obtains the information about CPUs from the operating system.

The above factors determine the default number of parallel execution servers to use. However, the actual number of processes used is limited by their availability on the requested instances during run time. The initialization parameter PARALLEL_MAX_SERVERS sets an upper limit on the total number of parallel execution servers that an instance can have.

If a minimum fraction of the desired parallel execution servers is not available (specified by the initialization parameter PARALLEL_MIN_PERCENT), a user error is produced. The user can then retry the query with less parallelism.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for information about adjusting the degree of parallelism

Adaptive Multi-User Algorithm

When the adaptive multi-user algorithm is enabled, the parallel execution coordinator varies the degree of parallelism according to the system load. The Database Resource Manager determines the load by calculating the number of allocated threads. If the number of threads currently allocated is larger than the optimal number of threads, given the number of available CPUs, the algorithm reduces the degree of parallelism. This reduction improves throughput by avoiding overallocation of resources.

Minimum Number of Parallel Execution Servers

Oracle can perform an operation in parallel as long as at least two parallel execution servers are available. If too few parallel execution servers are available, your SQL statement may execute slower than expected. You can specify that a minimum

percentage of requested parallel execution servers must be available in order for the operation to execute. This strategy ensures that your SQL statement executes with a minimum acceptable parallel performance. If the minimum percentage of requested parallel execution servers are not available, the SQL statement does not execute and returns an error.

The initialization parameter PARALLEL_MIN_PERCENT specifies the desired minimum percentage of requested parallel execution servers. This parameter affects DML and DDL operations as well as queries.

For example, if you specify 50 for this parameter, then at least 50% of the parallel execution servers requested for any parallel operation must be available in order for the operation to succeed. If 20 parallel execution servers are requested, then at least 10 must be available or an error is returned to the user. If PARALLEL_MIN_PERCENT is set to null, then all parallel operations will proceed as long as at least two parallel execution servers are available for processing.

Limiting the Number of Available Instances

In an Oracle Parallel Server, instance groups can be used to limit the number of instances that participate in a parallel operation. You can create any number of instance groups, each consisting of one or more instances. You can then specify which instance group is to be used for any or all parallel operations. Parallel execution servers will only be used on instances which are members of the specified instance group.

> **See Also:** *Oracle8i Parallel Server Concepts* for more information about instance groups

Balancing the Work Load

To optimize performance, all parallel execution servers should have equal work loads. For SQL statements parallelized by block range or by parallel execution servers, the work load is dynamically divided among the parallel execution servers. This minimizes *workload skewing*, which occurs when some parallel execution servers perform significantly more work than the other processes.

For SQL statements parallelized by partitions, if the work load is evenly distributed among the partitions then you can optimize performance by matching the number of parallel execution servers to the number of partitions, or by choosing a degree of parallelism such that the number of partitions is a multiple of the number of processes.

For example, if a table has ten partitions and a parallel operation divides the work evenly among them, you can use ten parallel execution servers (degree of parallelism = 10) to do the work in approximately one-tenth the time that one process would take, or you can use five processes to do the work in one-fifth the time, or two processes to do the work in one-half the time.

If, however, you use nine processes to work on ten partitions, the first process to finish its work on one partition then begins work on the tenth partition; and as the other processes finish their work they become idle. This does not give good performance when the work is evenly divided among partitions. When the work is unevenly divided, the performance varies depending on whether the partition that is left for last has more or less work than the other partitions.

Similarly, if you use four processes to work on ten partitions and the work is evenly divided, then each process works on a second partition after finishing its first partition, but only two of the processes work on a third partition while the other two remain idle.

In general, you cannot assume that the time taken to perform a parallel operation on N partitions with P parallel execution servers will be N / P, because of the possibility that some processes might have to wait while others finish working on the last partition(s). By choosing an appropriate degree of parallelism, however, you can minimize the workload skewing and optimize performance.

> **See Also:** "Affinity and Parallel DML" on page 23-50 for information about balancing the work load with disk affinity

Parallelization Rules for SQL Statements

A SQL statement can be parallelized if it includes a parallel hint or if the table or index being operated on has been declared PARALLEL with a CREATE or ALTER statement. In addition, a data definition language (DDL) statement can be parallelized by using the PARALLEL clause. However, not all of these methods apply to all types of SQL statements.

Parallelization has two components: the decision to parallelize and the degree of parallelism. These components are determined differently for queries, DDL operations, and DML operations.

To determine the degree of parallelism, Oracle looks at the *reference objects*:

- Parallel query looks at each table and index, in the portion of the query being parallelized, to determine which is the reference table. The basic rule is to pick the table or index with the largest degree of parallelism.

- For parallel DML (insert, update, and delete), the reference object that determines the degree of parallelism is the table being modified by an insert, update, or delete operation. Parallel DML also adds some limits to the degree of parallelism to prevent deadlock. If the parallel DML statement includes a subquery, the subquery's degree of parallelism is the same as the DML operation.
- For parallel DDL, the reference object that determines the degree of parallelism is the table, index, or partition being created, rebuilt, split, or moved. If the parallel DDL statement includes a subquery, the subquery's degree of parallelism is the same as the DDL operation.

Rules for Parallelizing Queries

Decision to Parallelize A SELECT statement can be parallelized only if the following conditions are satisfied:

1. The query includes a parallel hint specification (PARALLEL or PARALLEL_INDEX) or the schema objects referred to in the query have a PARALLEL declaration associated with them.
2. At least one of the tables specified in the query requires one of the following:
   - A full table scan
   - An index range scan spanning multiple partitions

Degree of Parallelism The degree of parallelism for a query is determined by the following rules:

1. The query uses the maximum degree of parallelism taken from all of the table declarations involved in the query and all of the potential indexes that are candidates to satisfy the query (the *reference objects*). That is, the table or index that has the greatest degree of parallelism determines the query's degree of parallelism (*maximum query directive*).
2. If a table has both a parallel hint specification in the query and a parallel declaration in its table specification, the hint specification takes precedence over parallel declaration specification.

Rules for Parallelizing UPDATE and DELETE

Update and delete operations are parallelized by partition or subpartition. Updates and deletes can only be parallelized on partitioned tables; update/delete parallelism is not possible within a partition, nor on a nonpartitioned table.

You have two ways to specify parallel directives for UPDATE and DELETE operations (assuming that PARALLEL DML mode is enabled):

1. Parallel clause specified in the definition of the table being updated or deleted (the reference object).
2. Update or delete parallel hint specified at the statement.

Parallel hints are placed immediately after the UPDATE or DELETE keywords in UPDATE and DELETE statements. The hint also applies to the underlying scan of the table being changed.

Parallel clauses in CREATE TABLE and ALTER TABLE statements specify table parallelism. If a parallel clause exists in a table definition, it determines the parallelism of DML statements as well as queries. If the DML statement contains explicit parallel hints for a table, however, then those hints override the effect of parallel clauses for that table.

You can use the ALTER SESSION FORCE PARALLEL DML statement to override parallel clauses for subsequent update and delete statements in a session. Parallel hints in update and delete statements override the ALTER SESSION FORCE PARALLEL DML statement.

> **See Also:** "Composite Partitioning" on page 11-16 for information about updating and deleting when there are subpartitions

Decision to Parallelize The following rule determines whether the update / delete operation should be parallelized in an update / delete statement:

The UPDATE or DELETE operation will be parallelized *if and only if* at least one of the following is true:

- The table being updated / deleted has a PARALLEL specification.
- The PARALLEL hint is specified in the DML statement.
- An ALTER SESSION FORCE PARALLEL DML statement was issued previously during the session.

If the statement contains subqueries or updatable views, then they may have their own separate parallel hints or clauses, but these parallel directives do not affect the decision to parallelize the update or delete.

Although the parallel hint or clause on the tables is used by both query and update / delete portions to determine parallelism, the decision to parallelize the update / delete portion is made independently of the query portion, and vice versa.

Degree of Parallelism The degree of parallelism is determined by the same rules as for the queries. Note that in the case of update and delete operations, only the target table to be modified (the only reference object) is involved.

The precedence rule to determine the degree of parallelism for the update / delete operation is that the update or delete parallel hint specification takes precedence over the parallel declaration specification of the target table:

Update / Delete hint > Parallel declaration specification of target table

The maximum degree of parallelism you can achieve is equal to the number of partitions (or subpartitions in the case of composite subpartitions) in the table. A parallel execution server can update into or delete from multiple partitions, but each partition can only be updated or deleted by one parallel execution server.

If the degree of parallelism is less than the number of partitions, then the first process to finish work on one partition continues working on another partition, and so on until the work is finished on all partitions. If the degree of parallelism is greater than the number of partitions involved in the operation, then the excess parallel execution servers would have no work to do.

Example 1: `UPDATE tbl_1 SET c1=c1+1 WHERE c1>100;`

If TBL_1 is a partitioned table and its table definition has a parallel clause, then the update operation will be parallelized even if the scan on the table is serial (such as an index scan), assuming that the table has more than one partition with C1 greater than 100.

Example 2: `UPDATE /*+ PARALLEL(tbl_2,4) */ tbl_2 SET c1=c1+1;`

Both the scan and update operations on TBL_2 will be parallelized with degree 4.

Rules for Parallelizing INSERT ... SELECT

An INSERT ... SELECT statement parallelizes its INSERT and SELECT operations independently, except for the degree of parallelism.

You can specify a parallel hint after the INSERT keyword in an INSERT ... SELECT statement. Because the tables being queried are usually not the same as the table being inserted into, the hint allows you to specify parallel directives specifically for the insert operation.

You have the following ways to specify parallel directives for an INSERT... SELECT statement (assuming that PARALLEL DML mode is enabled):

- SELECT parallel hint(s) specified at the statement.
- Parallel clause(s) specified in the definition of tables being selected.
- INSERT parallel hint specified at the statement.
- Parallel clause specified in the definition of tables being inserted into.

You can use the ALTER SESSION FORCE PARALLEL DML statement to override parallel clauses for subsequent insert operations in a session. Parallel hints in insert operations override the ALTER SESSION FORCE PARALLEL DML statement.

Decision to Parallelize The following rule determines whether the insert operation should be parallelized in an INSERT... SELECT statement:

> The INSERT operation will be parallelized *if and only if* at least one of the following is true:
>
> - The PARALLEL hint is specified after the INSERT in the DML statement.
> - The table being inserted into (the reference object) has a PARALLEL declaration specification.
> - An ALTER SESSION FORCE PARALLEL DML statement was issued previously during the session.

Hence the decision to parallelize the insert operation is made independently of the select operation, and vice versa.

Degree of Parallelism Once the *decision* to parallelize the select and / or insert operation is made, one parallel directive is picked for deciding *degree* of parallelism of the whole statement using the following precedence rule:

> Insert Hint directive > Parallel declaration specification of the inserting table > Maximum Query directive

where *Maximum Query directive* means that among multiple tables and indexes, the table or index that has the maximum degree of parallelism determines the parallelism for the query operation.

The chosen parallel directive is applied to both the select and insert operations.

Example: In the following example, the degree of parallelism used will be 2, which is the degree specified in the Insert hint:

```
INSERT /*+ PARALLEL(tbl_ins,2) */ INTO tbl_ins
  SELECT /*+ PARALLEL(tbl_sel,4) */ * FROM tbl_sel;
```

Rules for Parallelizing DDL Statements

Decision to Parallelize DDL operations can be parallelized if a PARALLEL clause *(declaration)* is specified in the syntax. In the case of CREATE INDEX and ALTER INDEX ... REBUILD or ALTER INDEX ... REBUILD PARTITION, the parallel declaration is stored in the data dictionary.

You can use the ALTER SESSION FORCE PARALLEL DDL statement to override the parallel clauses of subsequent DDL statements in a session.

Degree of Parallelism The degree of parallelism is determined by the specification in the PARALLEL clause, unless it is overridden by an ALTER SESSION FORCE PARALLEL DDL statement. A rebuild of a partitioned index is never parallelized.

Rules for Parallelizing Create Index, Rebuild Index, Move/Split Partition

Parallel CREATE INDEX or ALTER INDEX ... REBUILD The CREATE INDEX and ALTER INDEX ... REBUILD statements can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement.

ALTER INDEX ... REBUILD can be parallelized only for a nonpartitioned index, but ALTER INDEX ... REBUILD PARTITION can be parallelized by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement.

The scan operation for ALTER INDEX ... REBUILD (nonpartitioned), ALTER INDEX ... REBUILD PARTITION, and CREATE INDEX has the same parallelism as the REBUILD or CREATE operation and uses the same degree of parallelism. If the degree of parallelism is not specified for REBUILD or CREATE, the default is the number of CPUs.

Parallel MOVE PARTITION or SPLIT PARTITION The ALTER INDEX ... MOVE PARTITION and ALTER INDEX ... SPLIT PARTITION statements can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement. Their scan operations have the same parallelism as the corresponding MOVE or SPLIT operations. If the degree of parallelism is not specified, the default is the number of CPUs.

Rules for Parallelizing Create Table as Select

The CREATE TABLE ... AS SELECT statement contains two parts: a CREATE part (DDL) and a SELECT part (query). Oracle can parallelize both parts of the statement. The CREATE part follows the same rules as other DDL operations.

Decision to Parallelize (Query Part) The query part of a CREATE TABLE ... AS SELECT statement can be parallelized only if the following conditions are satisfied:

1. The query includes a parallel hint specification (PARALLEL or PARALLEL_INDEX) or the CREATE part of the statement has a PARALLEL clause specification or the schema objects referred to in the query have a PARALLEL declaration associated with them.
2. At least one of the tables specified in the query requires one of the following:
   - A full table scan
   - An index range scan spanning multiple partitions

Degree of Parallelism (Query Part) The degree of parallelism for the query part of a CREATE TABLE ... AS SELECT statement is determined by one of these rules:

- The query part uses the values specified in the PARALLEL clause of the CREATE part.
- If the PARALLEL clause is not specified, the default degree of parallelism is the number of CPUs.
- If the CREATE is serial, then the degree of parallelism is determined by the query.

Note that any values specified in a hint for parallelism are ignored.

See Also: "Rules for Parallelizing Queries" on page 23-21

Decision to Parallelize (Create Part) The CREATE operation of CREATE TABLE ... AS SELECT can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement.

When the CREATE operation of CREATE TABLE ... AS SELECT is parallelized, Oracle also parallelizes the scan operation if possible. The scan operation cannot be parallelized if, for example:

- The SELECT clause has a NOPARALLEL hint
- The operation scans an index of a nonpartitioned table

When the CREATE operation is not parallelized, the SELECT can be parallelized if it has a PARALLEL hint or if the selected table (or partitioned index) has a parallel declaration.

Degree of Parallelism (Create Part) The degree of parallelism for the CREATE operation, and for the SELECT operation if it is parallelized, is specified by the PARALLEL clause of the CREATE statement, unless it is overridden by an ALTER SESSION FORCE PARALLEL DDL statement. If the PARALLEL clause does not specify the degree of parallelism, the default is the number of CPUs.

Summary of Parallelization Rules

Table 23–1 shows how various types of SQL statements can be parallelized, and indicates which methods of specifying parallelism take precedence.

- The priority (1) specification overrides priority (2) and priority (3).
- The priority (2) specification overrides priority (3).

> **See Also:** *Oracle8i SQL Reference*. for details about parallel clauses and hints in SQL statements

Table 23–1 Parallelization Rules

Parallel Operation	Parallelized by Clause, Hint, or Underlying Table/Index Declaration (priority order: 1, 2, 3)			
	Parallel Hint	Parallel Clause	ALTER SESSION	Parallel Declaration
Parallel query table scan (partitioned or nonpartitioned table)	(1) PARALLEL		(2) FORCE PARALLEL QUERY	(3) of table
Parallel query index range scan (partitioned index)	(1) PARALLEL_ INDEX		(2) FORCE PARALLEL QUERY	(2) of index
Parallel UPDATE / DELETE (partitioned table only)	(1) PARALLEL		(2) FORCE PARALLEL DML	(3) of table being updated or deleted from
Insert operation of parallel INSERT... SELECT (partitioned or nonpartitioned table)	(1) PARALLEL of insert		(2) FORCE PARALLEL DML	(3) of table being inserted into
Select operation of INSERT ... SELECT when INSERT is parallel	takes degree from INSERT statement			
Select operation of INSERT ... SELECT when INSERT is serial	(1) PARALLEL			(2) of table being selected from

Table 23–1 Parallelization Rules (Cont.)

Parallel Operation	Parallelized by Clause, Hint, or Underlying Table/Index Declaration (priority order: 1, 2, 3)			
	Parallel Hint	**Parallel Clause**	**ALTER SESSION**	**Parallel Declaration**
Create operation of parallel CREATE TABLE ... AS SELECT (partitioned or nonpartitioned table)	(**Note:** Hint in select clause does not affect the create operation.)	(2)	(1) FORCE PARALLEL DDL	
Select operation of CREATE TABLE ... AS SELECT when CREATE is parallel	takes degree from CREATE statement			
Select operation of CREATE TABLE ... AS SELECT when CREATE is serial	(1) PARALLEL or PARALLEL_ INDEX			(2) of querying tables or partitioned indexes
Parallel CREATE INDEX (partitioned or nonpartitioned index)		(2)	(1) FORCE PARALLEL DDL	
Parallel REBUILD INDEX (nonpartitioned index)		(2)	(1) FORCE PARALLEL DDL	
REBUILD INDEX (partitioned index)—never parallelized			—	—
Parallel REBUILD INDEX partition		(2)	(1) FORCE PARALLEL DDL	
Parallel MOVE / SPLIT partition		(2)	(1) FORCE PARALLEL DDL	

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 executed serially without notification.

See Also:

- "Operations That Can Be Parallelized" on page 23-2 lists the query operations that Oracle can parallelize
- "How Oracle Parallelizes Operations" on page 23-3 describes dynamic parallelism by rowid range
- "Process Architecture for Parallel Execution" on page 23-5 describes the processes that perform parallel queries
- "Parallelizing SQL Statements" on page 23-10 explains how the processes perform parallel queries
- "Rules for Parallelizing Queries" on page 23-21 explains the conditions for parallelizing a query and the factors that determine the degree of parallelism
- "Distributed Transaction Restrictions" on page 23-47 for examples of queries that reference a remote object

Parallel Queries on Index-Organized Tables

The following parallel scan methods are supported on index-organized tables:

- Parallel fast full scan of a nonpartitioned index-organized table
- Parallel fast full scan of a partitioned index-organized table
- Parallel index range scan of a partitioned index-organized table

These scan methods can be used for index-organized tables with overflow areas and index-organized tables that contain LOBs.

Nonpartitioned Index-Organized Tables

Parallel query on a nonpartitioned index-organized table uses parallel fast full scan. The degree of parallelism is determined, in decreasing order of priority, by:

1. The PARALLEL hint (if present)
2. The parallel degree associated with the table if the parallel degree is specified in the CREATE TABLE or ALTER TABLE statement

The allocation of work is done by dividing the index segment into a sufficiently large number of block ranges and then assigning block ranges to parallel execution servers in a demand-driven manner. The overflow blocks corresponding to any row are accessed in a demand-driven manner only by the process which owns that row.

Partitioned Index-Organized Tables

Both index range scan and fast full scan can be performed in parallel. For parallel fast full scan, parallelization is exactly the same as for nonpartitioned index-organized tables. For parallel index range scan on partitioned index-organized tables, the degree of parallelism is the minimum of the degree picked up from the above priority list (like in parallel fast full scan) and the number of partitions in the index-organized table. Depending on the degree of parallelism, each parallel execution server gets one or more partitions (assigned in a demand-driven manner), each of which contains the primary key index segment and the associated overflow segment, if any.

Parallel Queries on Object Types

Parallel queries can be performed on object type tables and tables containing object type columns. Parallel query for object types supports all of the features that are available for sequential queries on object types, including:

- Methods on object types
- Attribute access of object types
- Constructors to create object type instances
- Object views
- PL/SQL and OCI queries for object types

There are no limitations on the size of the object types for parallel queries.

The following restrictions apply to using parallel query for object types.

- A MAP function is needed to parallelize queries involving joins and sorts (through ORDER BY, GROUP BY, or set operations). In the absence of a MAP function the query will automatically be executed serially.
- Parallel queries on nested tables are not supported. Even in the presence of a parallel attribute for the table or parallel hints, the query will execute serially.
- Parallel DML and parallel DDL are not supported with object types. DML and DDL statements are always performed serially.

In all cases where the query cannot execute in parallel because of any of the above restrictions, the whole query executes serially without giving an error message.

Parallel DDL

This section includes the following topics on parallelism for data definition language (DDL) statements:

DDL Statements That Can Be Parallelized

You can parallelize DDL statements for tables and indexes that are nonpartitioned or partitioned. Table 23–1 on page 23-27 summarizes the operations that can be parallelized in DDL statements.

The parallel DDL statements for nonpartitioned tables and indexes are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER INDEX ... REBUILD

The parallel DDL statements for partitioned tables and indexes are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER TABLE ... COALESCE PARTITION
- ALTER INDEX ... REBUILD PARTITION
- ALTER INDEX ... SPLIT PARTITION—only if the (global) index partition being split is Usable

All of these DDL operations can be performed in no-logging mode for either parallel or serial execution.

CREATE TABLE for an index-organized table can be parallelized either with or without an AS SELECT clause.

Different parallelism is used for different operations (see Table 23–1 on page 23-27). Parallel create (partitioned) table as select and parallel create (partitioned) index execute with a degree of parallelism equal to the number of partitions.

Partition parallel analyze table is made less necessary by the ANALYZE {TABLE, INDEX} PARTITION statements, since parallel analyze of an entire partitioned table can be constructed with multiple user sessions.

Parallel DDL cannot occur on tables with object columns or LOB columns.

See Also:

- "Logging Mode" on page 22-5
- *Oracle8i SQL Reference* for information about the syntax and use of parallel DDL statements
- *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for information about LOB restrictions

CREATE TABLE ... AS SELECT in Parallel

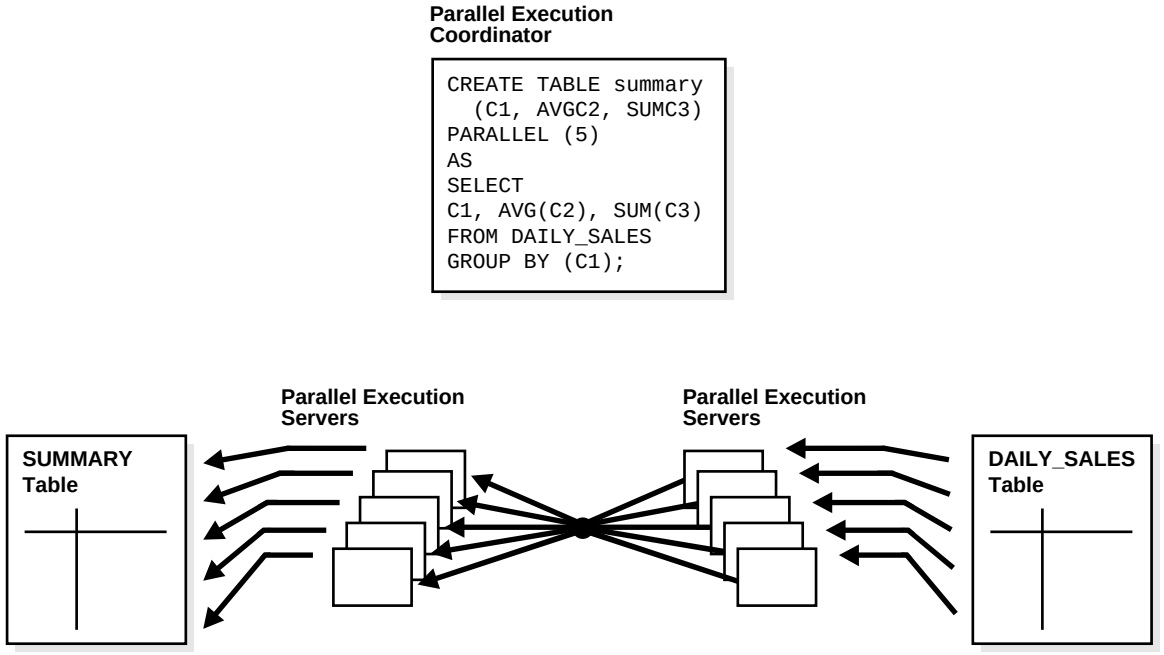
For performance reasons, decision support applications often require large amounts of data to be summarized or "rolled up" into smaller tables for use with ad hoc, decision support queries. Rollup occurs regularly (such as nightly or weekly) during a short period of system inactivity.

Parallel execution allows you to parallelize the query and create operations of creating a table as a subquery from another table or set of tables.

Figure 23–6 illustrates creating a table from a subquery in parallel.

Note: Clustered tables cannot be created and populated in parallel.

Figure 23–6 Creating a Summary Table in Parallel



Recoverability and Parallel DDL

When summary table data is derived from other tables' data, the recoverability from media failure for the smaller summary table may not be important and can be turned off during creation of the summary table.

If you disable logging during parallel table creation (or any other parallel DDL operation), you should take a backup of the tablespace containing the table once the table is created to avoid loss of the table due to media failure.

Use the NOLOGGING clause of CREATE / ALTER TABLE / INDEX statements to disable undo and redo log generation.

See Also:

- "Logging Mode" on page 22-5
- *Oracle8i Administrator's Guide* for information about recoverability of tables created in parallel

Space Management for Parallel DDL

Creating a table or index in parallel has space management implications that affect both the storage space required during the parallel operation and the free space available after the table or index has been created.

Storage Space for CREATE TABLE ... AS SELECT and CREATE INDEX

When creating a table or index in parallel, each parallel execution server uses the values in the STORAGE clause of the CREATE statement to create temporary segments to store the rows. Therefore, a table created with an INITIAL of 5M and a PARALLEL DEGREE of 12 consumes at least 60 megabytes (MB) of storage during table creation, because each process starts with an extent of 5 MB. When the parallel execution coordinator combines the segments, some of the segments may be trimmed, and the resulting table may be smaller than the requested 60 MB.

> **See Also:** *Oracle8i SQL Reference* for a discussion of the syntax of the CREATE TABLE statement

Free Space and Parallel DDL

When you create indexes and tables in parallel, each parallel execution server allocates a new extent and fills the extent with the table or index's data. Thus, if you create an index with a degree of parallelism of 3, there will be at least three extents for that index initially. This discussion also applies to rebuilding indexes in parallel and moving, splitting, or rebuilding partitions in parallel.

Serial operations require the schema object to have at least one extent. Parallel creations require that tables or indexes have at least as many extents as there are parallel execution servers creating the schema object.

When you create a table or index in parallel, it is possible to create pockets of free space—either external or internal fragmentation. This occurs when the temporary segments used by the parallel execution servers are larger than what is needed to store the rows.

- If the unused space in each temporary segment is larger than the value of the MINIMUM EXTENT parameter set at the tablespace level, then Oracle trims the unused space when merging rows from all of the temporary segments into the table or index. The unused space is returned to the system free space and can be allocated for new extents, but it cannot be coalesced into a larger segment because it is not contiguous space (*external fragmentation*).
- If the unused space in each temporary segment is smaller than the value of the MINIMUM EXTENT parameter, then unused space cannot be trimmed when

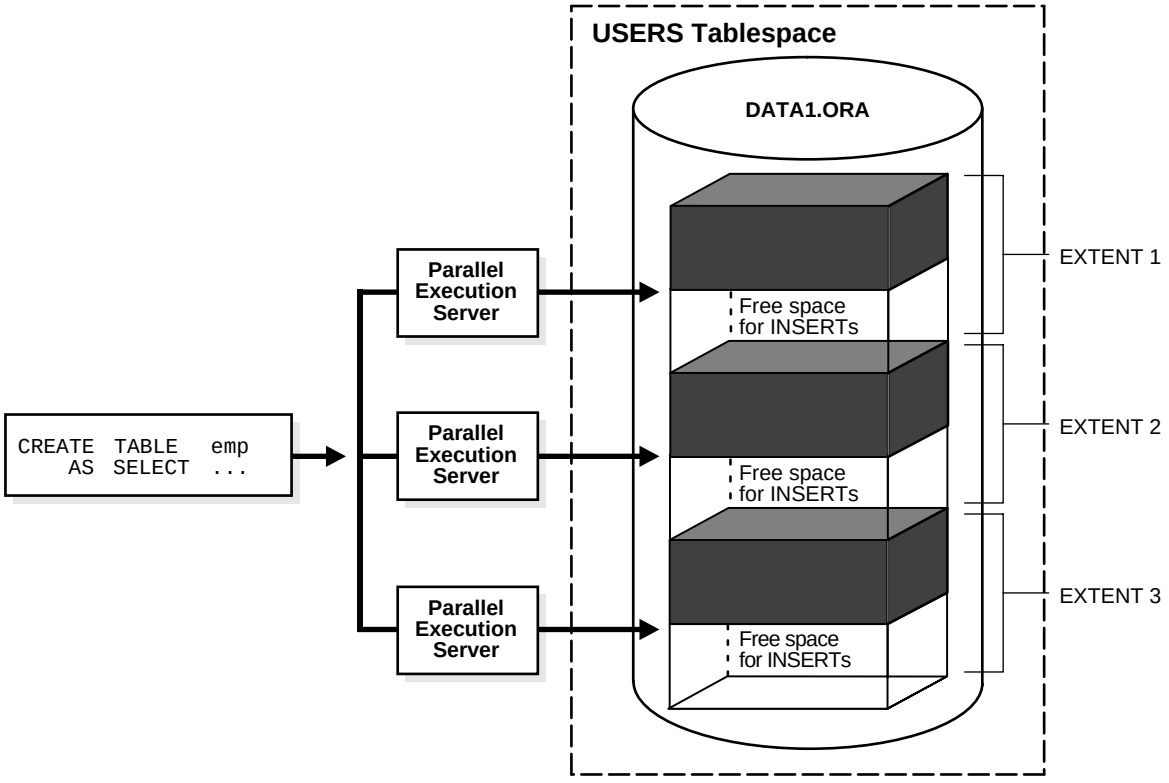
the rows in the temporary segments are merged into the table or index. This unused space is not returned to the system free space; it becomes part of the table or index (*internal fragmentation*) and is available only for subsequent inserts or for updates that require additional space.

For example, if you specify a degree of parallelism of three for a CREATE TABLE ... AS SELECT statement but there is only one datafile in the tablespace, then the internal fragmentation illustrated in Figure 23–7 can arise. The pockets of free space within internal table extents of a datafile cannot be coalesced with other free space and allocated as extents.

See Also:

- Chapter 3, "Tablespaces and Datafiles" for information about coalescing free space
- *Oracle8i Designing and Tuning for Performance* for more information about creating tables and indexes in parallel

Figure 23–7 Unusable Free Space (Internal Fragmentation)



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.

Note: Although generally data manipulation language (DML) includes queries, in this chapter the term "DML" refers only to inserts, updates, and deletes.

This section discusses the following parallel DML topics:

- Advantages of Parallel DML over Manual Parallelism
- When to Use Parallel DML
- Enabling Parallel DML
- Transaction Model for Parallel DML
- Recovery for Parallel DML
- Space Considerations for Parallel DML
- Lock and Enqueue Resources for Parallel DML
- Restrictions on Parallel DML

> **See Also:** Chapter 22, "Direct-Load INSERT" for a detailed description of parallel insert statements

Advantages of Parallel DML over Manual Parallelism

You can parallelize DML operations manually by issuing multiple DML statements simultaneously against different sets of data. For example, you can parallelize manually by:

- Issuing multiple INSERT statements to multiple instances of an Oracle Parallel Server to make use of free space from multiple free list blocks
- Issuing multiple UPDATE and DELETE statements with different key value ranges or rowid ranges.

However, manual parallelism has the following disadvantages:

- Difficult to use: you have to open multiple sessions (possibly on different instances) and issue multiple statements.
- Lack of transactional properties: the DML statements are issued at different times, thus the changes are done with inconsistent snapshots of the database. To get atomicity, the commit or rollback of the various statements must be coordinated manually (maybe across instances).
- Work division complexity: you may have to query the table in order to find out the rowid or key value ranges to correctly divide the work.
- Lack of affinity and resource information: you need to know affinity information to issue the right DML statement at the right instance when

running an Oracle Parallel Server. You also have to find out about current resource usage to balance work load across instances.

Parallel DML removes these disadvantages by performing inserts, updates, and deletes in parallel automatically.

When to Use Parallel DML

Parallel DML operations are mainly used to speed up large DML operations against large database objects. Parallel DML is useful in a decision support system (DSS) environment where the performance and scalability of accessing large objects are important. Parallel DML complements parallel query in providing you with both querying and updating capabilities for your DSS databases.

The overhead of setting up parallelism makes parallel DML operations infeasible for short OLTP transactions. However, parallel DML operations can speed up batch jobs running in an OLTP database.

Refresh Tables of a Data Warehouse System

In a data warehouse system, large tables need to be *refreshed* (updated) periodically with new or modified data from the production system. You can do this efficiently by using parallel DML combined with updatable join views.

The data that needs to be refreshed is generally loaded into a temporary table before starting the refresh process. This table contains either new rows or rows that have been updated since the last refresh of the data warehouse. You can use an updatable join view with parallel UPDATE to refresh the updated rows, and you can use an anti-hash join with parallel INSERT to refresh the new rows.

See Also: *Oracle8i Designing and Tuning for Performance*.

Intermediate Summary Tables

In a DSS environment, many applications require complex computations that involve constructing and manipulating many large intermediate summary tables. These summary tables are often temporary and frequently do not need to be logged. Parallel DML can speed up the operations against these large intermediate tables. One benefit is that you can put incremental results in the intermediate tables and perform parallel UPDATEs.

In addition, the summary tables may contain cumulative or comparison information which has to persist beyond application sessions; thus, temporary

tables are not feasible. Parallel DML operations can speed up the changes to these large summary tables.

Scoring Tables

Many DSS applications score customers periodically based on a set of criteria. The scores are usually stored in large DSS tables. The score information is then used in making a decision, for example, inclusion in a mailing list.

This scoring activity queries and updates a large number of rows in the large table. Parallel DML can speed up the operations against these large tables.

Historical Tables

Historical tables describe the business transactions of an enterprise over a recent time interval. Periodically, the DBA deletes the set of oldest rows and inserts a set of new rows into the table. Parallel INSERT... SELECT and parallel DELETE operations can speed up this rollover task.

Although you can also use parallel direct loader (SQL*Loader) to insert bulk data from an external source, parallel INSERT... SELECT will be faster in inserting data that already exists in another table in the database.

Dropping a partition can also be used to delete old rows, but to do this, the table has to be partitioned by date and with the appropriate time interval.

Batch Jobs

Batch jobs executed in an OLTP database during off hours have a fixed time window in which the jobs must complete. A good way to ensure timely job completion is to parallelize their operations. As the work load increases, more machine resources can be added; the scaleup property of parallel operations ensures that the time constraint can be met.

Enabling Parallel DML

A DML statement can be parallelized only if you have explicitly enabled parallel DML in the session via the ENABLE PARALLEL DML clause of the ALTER SESSION statement. This mode is required because parallel DML and serial DML have different locking, transaction, and disk space requirements.

The default mode of a session is DISABLE PARALLEL DML. When PARALLEL DML is disabled, no DML will be executed in parallel even if the PARALLEL hint or PARALLEL clause is used.

When PARALLEL DML is enabled in a session, all DML statements in this session will be *considered* for parallel execution. However, even if the PARALLEL DML is enabled, the DML operation may still execute serially if there are no parallel hints or parallel clauses or if restrictions on parallel operations are violated.

The session's PARALLEL DML mode does not influence the parallelism of SELECT statements, DDL statements, and the query portions of DML statements. Thus, if this mode is not set, the DML operation is not parallelized but scans or join operations within the DML statement may still be parallelized.

See Also:

- "Space Considerations for Parallel DML" on page 23-42
- "Lock and Enqueue Resources for Parallel DML" on page 23-43
- "Restrictions on Parallel DML" on page 23-44

Transactions with PARALLEL DML Enabled

A session that is enabled for PARALLEL DML may put transactions in the session in a special mode: If any DML statement in a transaction modifies a table in parallel, no subsequent *serial or parallel* query or DML statement can access the same table again in that transaction. This means that the results of parallel modifications cannot be seen during the transaction.

Serial or parallel statements that attempt to access a table which has already been modified in parallel within the same transaction are rejected with an error message.

If a PL/SQL procedure or block is executed in a PARALLEL DML enabled session, then this rule applies to statements in the procedure or block.

Transaction Model for Parallel DML

To execute a DML operation in parallel, the parallel execution coordinator acquires or spawns parallel execution servers and each parallel execution server executes a portion of the work under its own parallel process transaction.

- Each parallel execution server creates a different parallel process transaction.
- To reduce contention on the rollback segments, only a few parallel process transactions should reside in the same rollback segment (see the next section).

The coordinator also has its own coordinator transaction, which can have its own rollback segment.

Rollback Segments

Oracle assigns transactions to rollback segments that have the fewest active transactions. To speed up both forward and undo operations, you should create and bring online enough rollback segments so that at most two parallel process transactions are assigned to one rollback segment.

Create the rollback segments in tablespaces that have enough space for them to extend when necessary and set the MAXEXTENTS storage parameters for the rollback segments to UNLIMITED. Also, set the OPTIMAL value for the rollback segments so that after the parallel DML transactions commit, the rollback segments will be shrunk to the OPTIMAL size.

Two-Phase Commit

A parallel DML operation is executed by more than one independent parallel process transaction. In order to ensure user-level transactional atomicity, the coordinator uses a two-phase commit protocol to commit the changes performed by the parallel process transactions.

This two-phase commit protocol is a simplified version which makes use of shared disk architecture to speed up transaction status lookups, especially during transactional recovery. It does not require the Oracle XA library. In-doubt transactions never become visible to users.

Recovery for Parallel DML

The time required to roll back a parallel DML operation is roughly equal to the time it took to perform the forward operation.

Oracle supports parallel rollback after transaction and process failures, and after instance and system failures. Oracle can parallelize both the rolling forward stage and the rolling back stage of transaction recovery.

> **See Also:** *Oracle8i Backup and Recovery Guide* for details about parallel rollback

Transaction Recovery for User-Issued Rollback

A user-issued rollback in a transaction failure due to statement error is performed in parallel by the parallel execution coordinator and the parallel execution servers. The rollback takes approximately the same amount of time as the forward transaction.

Process Recovery

Recovery from the failure of a parallel DML coordinator or parallel execution server is performed by the PMON process. PMON performs the following tasks:

- If a single parallel execution server fails, PMON rolls back that process's work and all other parallel execution servers roll back their own work.
- If multiple parallel execution servers fail, PMON rolls back all of their work serially.
- If the parallel execution coordinator fails, PMON recovers the coordinator and all parallel execution servers roll back their own work in parallel.

System Recovery

Recovery from a system failure needs a new startup. Recovery is performed by the SMON process and any recovery server processes spawned by SMON. Parallel DML statements may be recovered in parallel using parallel rollback. If the initialization parameter COMPATIBLE is set to 8.1.3 or greater, *Fast-Start On-Demand Rollback* enables dead transactions to be recovered, on demand, one block at a time.

See Also: "Fast-Start On-Demand Rollback" on page 29-15

Instance Recovery (Oracle Parallel Server)

Recovery from an instance failure in an Oracle Parallel Server is performed by the recovery processes (that is, the SMON processes and any recovery server processes they spawn) of other live instances. Each recovery process of the live instances can recover the parallel execution coordinator and/or parallel execution server transactions of the failed instance independently.

Space Considerations for Parallel DML

Parallel UPDATE uses the space in the existing object, as opposed to direct-load INSERT which gets new segments for the data.

Space usage characteristics may be different in parallel than they would be if the statement executed sequentially, because multiple concurrent child transactions modify the object.

See Also: "Space Considerations" on page 22-9 for information about space for direct-load INSERT

Lock and Enqueue Resources for Parallel DML

A parallel DML operation's lock and enqueue resource requirements are very different from the serial DML requirements. Parallel DML holds many more locks, so you should increase the value of the ENQUEUE_RESOURCES and DML_LOCKS parameters.

The processes for a parallel UPDATE, DELETE, or INSERT statement acquire the following locks:

- The parallel execution coordinator acquires:
  - 1 table lock SX
  - 1 partition lock X per partition / subpartition

  For parallel INSERT into a partitioned table, the coordinator acquires partition locks for all partitions. For parallel UPDATE or DELETE, the coordinator acquires partition locks for all partitions, unless the WHERE clause limits the partitions involved.
- Each parallel execution server acquires:
  - 1 table lock SX
  - 1 partition lock NULL per partition / subpartition
  - 1 partition-wait lock X per partition / subpartition

A parallel execution server can work on one or more partitions, but a partition can only be worked on by one parallel execution server.

For example, for a table with 600 partitions running with parallel degree 100, a parallel DML statement needs the following locks (assuming all partitions are involved in the statement):

- The coordinator acquires 1 table lock SX and 600 partition locks X.
- Total parallel execution servers acquire 100 table locks SX, 600 partition locks NULL, and 600 partition-wait locks X.

A special type of parallel UPDATE exists called row-migrating parallel UPDATE. This parallel update method is only used when a table is defined with the row movement clause enabled and it allows rows to be moved to different partitions or subpartitions.

Table 23–2 summarizes the types of locks acquired by coordinator and parallel execution servers for different types of parallel DML statements.

Table 23–2 Locks Acquired by Parallel DML Statements

Type of statement	Parallel execution coordinator acquires:	Each parallel execution server acquires:
Parallel UPDATE or DELETE into partitioned table; WHERE clause pruned to a subset of partitions/subpartitions	1 table lock SX	1 table lock SX
	1 partition lock X per pruned (sub)partition	1 partition lock NULL per pruned (sub)partition owned by the parallel execution server
		1 partition-wait lock S per pruned (sub)partition owned by the parallel execution server
Parallel row-migrating UPDATE into partitioned table; WHERE clause pruned to a subset of (sub)partitions	1 table lock SX	1 table lock SX
	1 partition X lock per pruned (sub)partition	1 partition lock NULL per pruned (sub)partition owned by the parallel execution server
		1 partition-wait lock S per pruned partition owned by the parallel execution server
	1 partition lock SX for all other (sub)partitions	1 partition lock SX for all other (sub)partitions
Parallel UPDATE, DELETE, or INSERT into partitioned table	1 table lock SX	1 table lock SX
	Partition locks X for all (sub)partitions	1 partition lock NULL per (sub)partition owned by the parallel execution server
		1 partition-wait lock S per (sub)partition owned by the parallel execution server
Parallel INSERT into nonpartitioned table	1 table lock X	None

Restrictions on Parallel DML

The following restrictions apply to parallel DML (including direct-load INSERT):

- Update and delete operations are not parallelized on nonpartitioned tables.
- A transaction can contain multiple parallel DML statements that modify different tables, but after a parallel DML statement modifies a table, no subsequent *serial or parallel* statement (DML or query) can access the same table again in that transaction.

  - This restriction also exists after a serial direct-load INSERT statement: no subsequent SQL statement (DML or query) can access the modified table during that transaction.
  - Queries that access the same table are allowed before a parallel DML or direct-load INSERT statement, but not after.
  - Any serial or parallel statements attempting to access a table that has already been modified by a parallel UPDATE, parallel DELETE, or direct-load INSERT during the same transaction are rejected with an error message.

- If initialization parameter ROW_LOCKING = INTENT, then inserts, updates, and deletes are *not* parallelized (regardless of the serializable mode).
- Triggers are not supported for parallel DML operations.
- Replication functionality is not supported for parallel DML.
- Parallel DML cannot occur in the presence of certain constraints: self-referential integrity, delete cascade, and deferred integrity. In addition, for direct-load INSERT there is no support for any referential integrity.
- Parallel DML cannot occur on tables with object columns or LOB columns, or on index-organized tables.
- A transaction involved in a parallel DML operation cannot be or become a distributed transaction.
- Clustered tables are not supported.

Violations will cause the statement to execute serially without warnings or error messages (except for the restriction on statements accessing the same table in a transaction, which can cause error messages). For example, an update will be serialized if it is on a nonpartitioned table.

The following sections give further details about restrictions.

> **See Also:** *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for more information about LOB restrictions

Partitioning Key Restriction

You can only update the partitioning key of a partitioned table to a new value if the update would not cause the row to move to a new partition unless the table is defined with the row movement clause enabled.

Function Restrictions

The function restrictions for parallel DML are the same as those for parallel DDL and parallel query.

See Also: "Parallel Execution of Functions" on page 23-47

Data Integrity Restrictions

This section describes the interactions of integrity constraints and parallel DML statements.

NOT NULL and CHECK These types of integrity constraints are allowed. They are not a problem for parallel DML because they are enforced on the column and row level, respectively.

UNIQUE and PRIMARY KEY These types of integrity constraints are allowed.

FOREIGN KEY (Referential Integrity) There are restrictions for referential integrity whenever a DML operation on one table could cause a recursive DML operation on another table or, in order to perform the integrity check, it would be necessary to see simultaneously all changes made to the object being modified.

Table 23–3 lists all of the operations that are possible on tables that are involved in referential integrity constraints.

Table 23–3 Referential Integrity Restrictions

DML Statement	Issued on Parent	Issued on Child	Self-Referential
INSERT	(Not applicable)	Not parallelized	Not parallelized
UPDATE No Action	Supported	Supported	Not parallelized
DELETE No Action	Supported	Supported	Not parallelized
DELETE Cascade	Not parallelized	(Not applicable)	Not parallelized

Delete Cascade *Delete on tables having a foreign key with delete cascade is not parallelized* because parallel execution servers will try to delete rows from multiple partitions (parent and child tables).

Self-Referential Integrity *DML on tables with self-referential integrity constraints is not parallelized if the referenced keys (primary keys) are involved.* For DML on all other columns, parallelism is possible.

Deferrable Integrity Constraints If there are any deferrable constraints on the table being operated on, the DML operation will not be parallelized.

Trigger Restrictions

A DML operation will not be parallelized if any triggers are enabled on the affected tables that may get fired as a result of the statement. This implies that DML statements on tables that are being replicated will not be parallelized.

Relevant triggers must be disabled in order to parallelize DML on the table. Note that enabling / disabling triggers invalidates dependent shared cursors.

Distributed Transaction Restrictions

A DML operation cannot be parallelized if it is in a distributed transaction or if the DML or the query operation is against a remote object.

Example 1: DML statement which queries a remote object:

```
INSERT /* APPEND PARALLEL (t3,2) */ INTO t3 SELECT * FROM t4@dblink;
```

The query operation is executed serially without notification because it references a remote object.

Example 2: DML operation on a remote object:

```
DELETE /*+ PARALLEL (t1, 2) */ FROM t1@dblink;
```

The DELETE operation is not parallelized because it references a remote object.

Example 3: In a distributed transaction:

```
SELECT * FROM t1@dblink;
DELETE /*+ PARALLEL (t2,2) */ FROM t2;
COMMIT;
```

The DELETE operation is not parallelized because it occurs in a distributed transaction (which is started by the SELECT statement).

Parallel Execution of Functions

The execution of user-written functions written in PL / SQL, in Java, or as external procedures in C, can be parallelized. Any PL / SQL package variables or Java static attributes used by the function are entirely private to each individual parallel

execution process, however, and are newly initialized at the start of each parallel execution process rather than being copied from the original session. Because of this, not all functions will generate correct results if executed in parallel.

To allow a user-written function to be executed in parallel, use the PARALLEL_ENABLE keyword when you declare the function in either the CREATE FUNCTION or CREATE PACKAGE statement.

Functions in Parallel Queries

In a SELECT statement or a subquery in a DML or DDL statement, a user-written function may be executed in parallel if it has been declared with the PARALLEL_ENABLE keyword, if it is declared in a package or type and has a PRAGMA RESTRICT_REFERENCES that indicates all of WNDS, RNPS, and WNPS, or if it is declared with CREATE FUNCTION and the system can analyze the body of the PL/SQL code and determine that the code neither writes to the database nor reads nor modifies package variables.

Other parts of a query or subquery can sometimes execute in parallel even if a given function execution must remain serial.

See Also:

- *Oracle8i Application Developer's Guide - Fundamentals* for the description of the pragma RESTRICT_REFERENCES
- *Oracle8i SQL Reference*. for the description of CREATE FUNCTION

Functions in Parallel DML and DDL Statements

In a parallel DML or DDL statement, as in a parallel query, a user-written function may be executed in parallel if it has been declared with the PARALLEL_ENABLE keyword, if it is declared in a package or type and has a PRAGMA RESTRICT_REFERENCES that indicates all of RNDS, WNDS, RNPS, and WNPS, or if it is declared with CREATE FUNCTION and the system can analyze the body of the PL/SQL code and determine that the code neither reads nor writes to the database nor reads nor modifies package variables.

For a parallel DML statement, any function call that cannot be executed in parallel causes the entire DML statement to be executed serially.

For an INSERT ... SELECT or CREATE TABLE ... AS SELECT statement, function calls in the query portion are parallelized according to the parallel query rules in the

prior paragraph; the query may be parallelized even if the remainder of the statement must execute serially, or vice versa.

Affinity

> **Note:** The features described in this section are available only if you have purchased Oracle8*i* Enterprise Edition with the Parallel Server Option. See *Getting to Know Oracle8i* for information about the features and options available with Oracle8*i* Enterprise Edition.

In a shared-disk cluster or massively parallel processing (MPP) configuration, an instance of the Oracle Parallel Server is said to have *affinity* for a device if the device is directly accessed from the processor(s) on which the instance is running. Similarly, an instance has *affinity* for a file if it has affinity for the device(s) that the file is stored on.

Determination of affinity may involve arbitrary determinations for files that are striped across multiple devices. Somewhat arbitrarily, an instance is said to have affinity for a tablespace (or a partition of a table or index within a tablespace) if the instance has affinity for the first file in the tablespace.

Oracle considers affinity when allocating work to parallel execution servers. The use of affinity for parallel execution of SQL statements is transparent to users.

Affinity and Parallel Queries

Affinity in parallel queries increases the speed of scanning data from disk by doing the scans on a processor that is near the data. This can provide a substantial performance increase for machines that do not naturally support shared disks.

The most common use of affinity is for a table or index partition to be stored in one file on one device. This configuration provides the highest availability by limiting the damage done by a device failure and makes best use of partition-parallel index scans.

DSS customers might prefer to stripe table partitions over multiple devices (probably a subset of the total number of devices). This allows some queries to prune the total amount of data being accessed using partitioning criteria and still obtain parallelism through rowid-range parallel table (partition) scans. If the devices are configured as a RAID, availability can still be very good. Even when used for DSS, indexes should probably be partitioned on individual devices.

Other configurations (for example, multiple partitions in one file striped over multiple devices) will yield correct query results, but you may need to use hints or explicitly set object attributes to select the correct degree of parallelism.

Affinity and Parallel DML

For parallel DML (inserts, updates, and deletes), affinity enhancements improve cache performance by routing the DML operation to the node that has affinity for the partition.

Affinity determines how to distribute the work among the set of instances and/or parallel execution servers to perform the DML operation in parallel. Affinity can improve performance of queries in several ways:

- For certain MPP architectures, Oracle uses device-to-node affinity information to determine on which nodes to spawn parallel execution servers *(parallel process allocation)* and which work granules (rowid ranges or partitions) to send to particular nodes *(work assignment)*. Better performance is achieved by having nodes mainly access local devices, giving a better buffer cache hit ratio for every node and reducing the network overhead and I/O latency.
- For SMP shared disk clusters, Oracle uses a round-robin mechanism to assign devices to nodes. Similar to item 1, this device-to-node affinity is used in determining parallel process allocation and work assignment.
- For SMP, cluster, and MPP architectures, process-to-device affinity is used to achieve device isolation. This reduces the chances of having multiple parallel execution servers accessing the same device simultaneously. This process-to-device affinity information is also used in implementing stealing between processes.

For partitioned tables and indexes, partition-to-node affinity information determines process allocation and work assignment. For shared-nothing MPP systems, the Oracle Parallel Server tries to assign partitions to instances taking the disk affinity of the partitions into account. For shared-disk MPP and cluster systems, partitions are assigned to instances in a round-robin manner.

Affinity is only available for parallel DML when running in an Oracle Parallel Server configuration. Affinity information which persists across statements will improve buffer cache hit ratios and reduce block pings between instances.

> **See Also:** *Oracle8i Parallel Server Administration, Deployment, and Performance*

Other Types of Parallelism

In addition to parallel SQL execution, Oracle can use parallelism for the following types of operations:

- Parallel recovery
- Parallel propagation (replication)
- Parallel load (the SQL*Loader utility)

Like parallel SQL, parallel recovery and parallel propagation are executed by a parallel execution coordinator and multiple parallel execution servers. Parallel load, however, uses a different mechanism.

The behavior of the parallel execution coordinator and parallel execution servers may differ, depending on what kind of operation they perform (SQL, recovery, or propagation). For example, if all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started:

- In the parallel SQL role, the parallel execution coordinator switches to serial processing
- In the parallel propagation role, the parallel execution coordinator returns an error

For a given session, the parallel execution coordinator coordinates only one kind of operation. A parallel execution coordinator cannot coordinate, for example, parallel SQL and parallel propagation or parallel recovery at the same time.

See Also:

- *Oracle8i Utilities* for information about parallel load and general information about SQL*Loader
- *Oracle8i Designing and Tuning for Performance* for advice about using parallel load
- "Performing Recovery in Parallel" on page 29-11
- *Oracle8i Backup and Recovery Guide* for detailed information about parallel recovery
- *Oracle8i Replication* for information about parallel propagation

Part VIII

Data Protection

Part VIII describes how Oracle protects the data in a database and explains what the database administrator can do to provide additional protection for data.

Part VIII contains the following chapters:

24

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

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 25, "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 24–1:

Table 24–1 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 and was the only automatic isolation level provided before Oracle Release 7.3.

> **See Also:** "How Oracle Manages Data Concurrency and Consistency" on page 24-4 for a full discussion of read committed and serializable isolation levels

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 24-15 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:

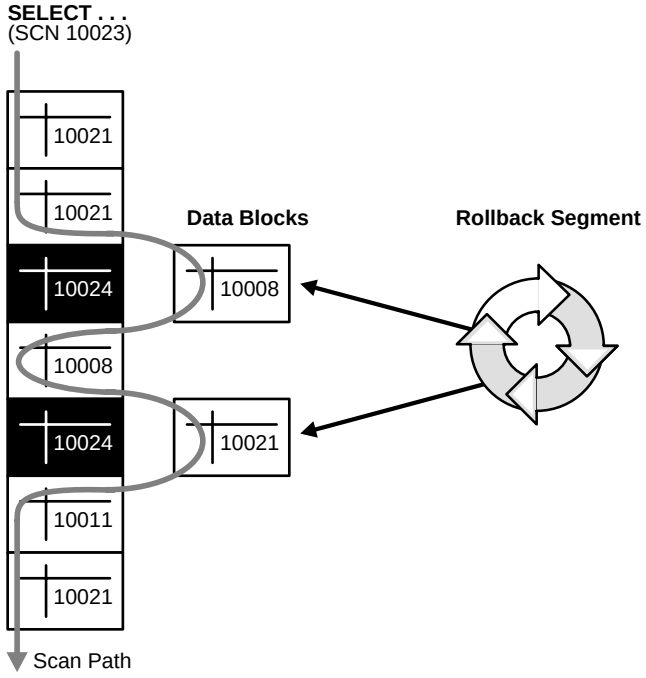
- Multiversion Concurrency Control
- Statement-Level Read Consistency
- Transaction-Level Read Consistency
- Read Consistency in the Oracle Parallel Server
- Oracle Isolation Levels
- Comparing Read Committed and Serializable Isolation
- Choosing an Isolation Level

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 24–1 shows how Oracle provides statement-level read consistency using data in rollback segments.

Figure 24–1 Transactions and Read Consistency



As a query enters the execution stage, the current system change number (SCN) is determined; in Figure 24–1, 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.

The "Snapshot Too Old" Message

In rare situations, Oracle cannot return a consistent set of results (often called a *snapshot*) for a long-running query. This occurs because not enough information remains in the rollback segments to reconstruct the older data. Usually, this error is produced when a lot of update activity causes the rollback segment to wrap around

and overwrite changes needed to reconstruct data that the long-running query requires. In this event, error 1555 will result:

```
ORA-1555: snapshot too old (rollback segment too small)
```

You can avoid this error by creating more or larger rollback segments. Alternatively, long-running queries can be issued when there are few concurrent transactions, or you can obtain a shared lock on the table you are querying, thus prohibiting any other exclusive locks during the transaction.

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 may 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 (see below), 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 in the Oracle Parallel Server

Oracle Parallel Server uses a parallel cache management technique called *Cache Fusion* to ensure data consistency among multiple instances that access a single database. When an inter-instance request for a consistent-read block creates a reader/writer cache coherency conflict, Cache Fusion uses the Block Server Process to copy blocks directly from the holding instance's memory cache to the requesting instance's memory cache. The instance holding the block rolls back uncommitted transactions and sends the block directly to the requestor without writing the block to disk. The state of the block is consistent as of the time at which the request was submitted at the requester node.

See Also:

- *Oracle8i Parallel Server Concepts*
- *Oracle8i Parallel Server Administration, Deployment, and Performance*

for details about Cache Fusion and the Block Server Process process

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 will never read dirty (uncommitted) data. Because Oracle does not prevent other transactions from modifying the data read by a query, that data may be changed by other transactions between two executions of the query. Thus, a transaction that executes a given query twice may experience both nonrepeatable read and phantoms.
serializable transactions	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.

Setting 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:** *Oracle8i 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 snapshot 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 may 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 will enable 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:

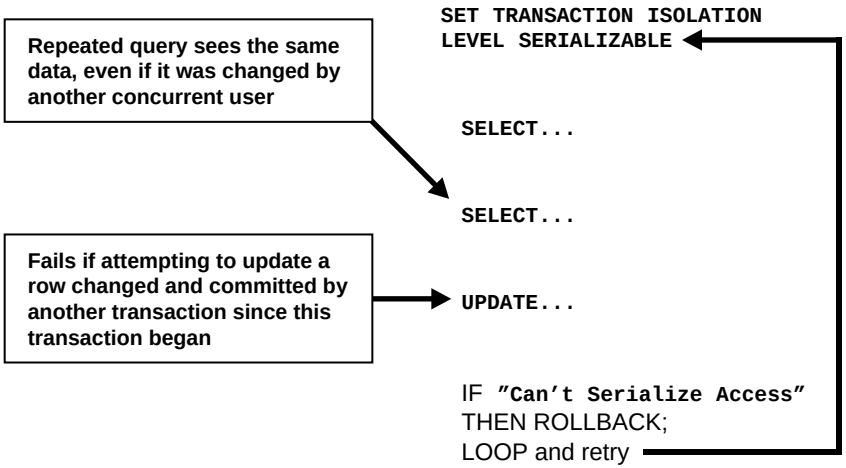
```
ORA-08177: Cannot serialize access for this transaction
```

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 24–2 shows an example of an application that rolls back and retries the transaction after it fails with the "Cannot serialize access" error:

Figure 24–2 Serializable Transaction Failure



Comparing 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 24–2 summarizes key differences between read committed and serializable transactions in Oracle.

Table 24–2 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 snapshot time	Statement	Transaction
Transaction set consistency	Statement level	Transaction level
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes
Waits for blocking transaction	Yes	Yes
Subject to "cannot serialize access"	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 should not 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:** *Oracle8i Application Developer's Guide - Fundamentals* for more information about referential integrity and serializable transactions

Oracle Parallel Server

You can use both read committed and serializable transaction isolation levels in an Oracle Parallel Server. An Oracle Parallel Server environment includes several Oracle instances running against a single database.

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:** *Oracle8i Distributed Database Systems*

Choosing an 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.

Both 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.

Choosing Read Committed Isolation

For many applications, read committed is the most appropriate isolation level. This is the isolation level used by applications running on Oracle releases before Release 7.3.

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 fe 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 reissue 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.

Choosing Serializable Isolation

Oracle's serializable isolation is suitable for environments where there is relatively low chance that two concurrent transactions will modify the same rows and the relatively 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 may 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 may 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.

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 24-19

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 24-32

Locking Modes

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 database will 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.

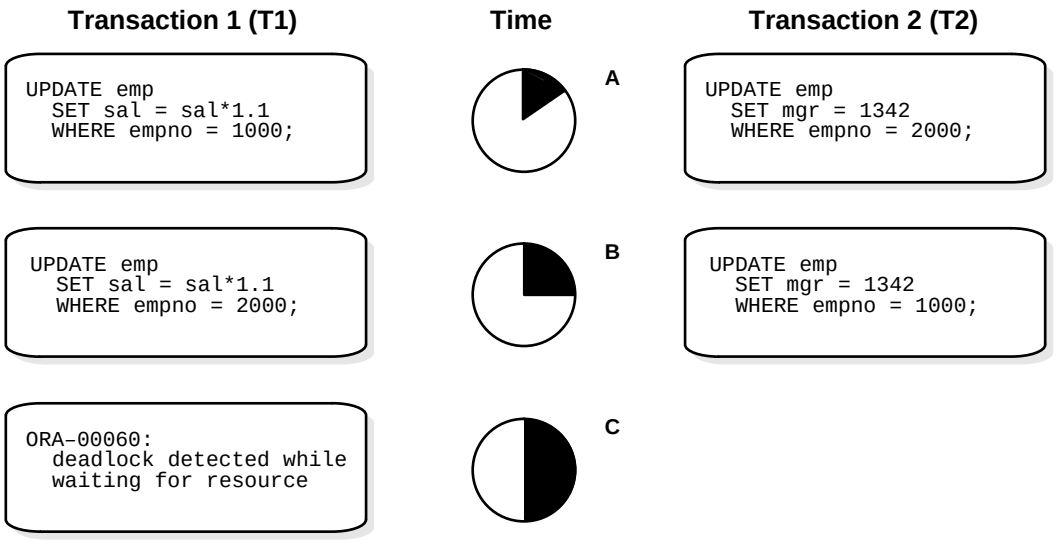
See Also: "Table Locks (TM)" on page 24-22

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 24–3 illustrates two transactions in a deadlock.

In Figure 24–3, 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 24–3 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 24-32 for more information about manually acquiring locks and for an example of a deadlock situation

Avoiding 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, you should 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.
distributed locks	Distributed locks ensure that the data and other resources distributed among the various instances of an Oracle Parallel Server remain consistent. Distributed locks are held by instances rather than transactions. They communicate the current status of a resource among the instances of an Oracle Parallel Server.
parallel cache management (PCM) locks	Parallel cache management locks are distributed locks that cover one or more data blocks (table or index blocks) in the buffer cache. PCM locks do not lock any rows on behalf of transactions.

This chapter discusses DML locks, DDL locks, and internal locks, respectively.

See Also:

- *Oracle8i Parallel Server Concepts*
- *Oracle8i Parallel Server Administration, Deployment, and Performance*

for more information about distributed locks and PCM 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. The following sections explain row and table locks.

> **Note:** The acronym in parentheses after each type of lock or lock mode in the following sections 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 above.

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:

- "Database Instance Failure" on page 29-4.
- "DDL Locks" on page 24-29

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 24–3 shows the table lock modes that statements acquire and operations that those locks permit and prohibit.

Table 24–3 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. Each section describes the mode of table lock, 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 24-32

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 24–4 summarizes the information in the following sections.

Table 24–4 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 20, "Oracle Dependency Management"

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 snapshots in the cluster. A DDL operation on a table or snapshot in a cluster acquires a share lock on the cluster, in addition to a share or exclusive DDL lock on the table or snapshot. 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 information will help you 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.

File and log locks are of particular importance if you are using the Oracle Parallel Server.

> **See Also:** *Oracle8i Parallel Server Administration, Deployment, and Performance* for more information about locks

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: *Oracle8i 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	

Transaction 1	Time Point	Transaction 2
	6	**ROLLBACK;** *(releases row locks)*
1 row processed. **ROLLBACK;**	7	
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)*

Transaction 1	Time Point	Transaction 2
ROLLBACK;	16	
	17	1 row processed. *(conflicting locks were released)* **ROLLBACK;**
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)*

Transaction 1	Time Point	Transaction 2
ROLLBACK;	25	
	26	1 row processed. *(conflicting table lock released)* **ROLLBACK;**
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.

Transcription 1	Time Point	Transaction 2
	34	**SELECT loc** **FROM scott.dept** **WHERE deptno = 20** **FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.
	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

Transaction 1	Time Point	Transaction 2
	44	**LOCK TABLE scott.dept** **IN ROW SHARE MODE** **NOWAIT;** ORA-00054
	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:** *Oracle8i Application Developer's Guide - Fundamentals* for more information about Oracle Lock Management services

25

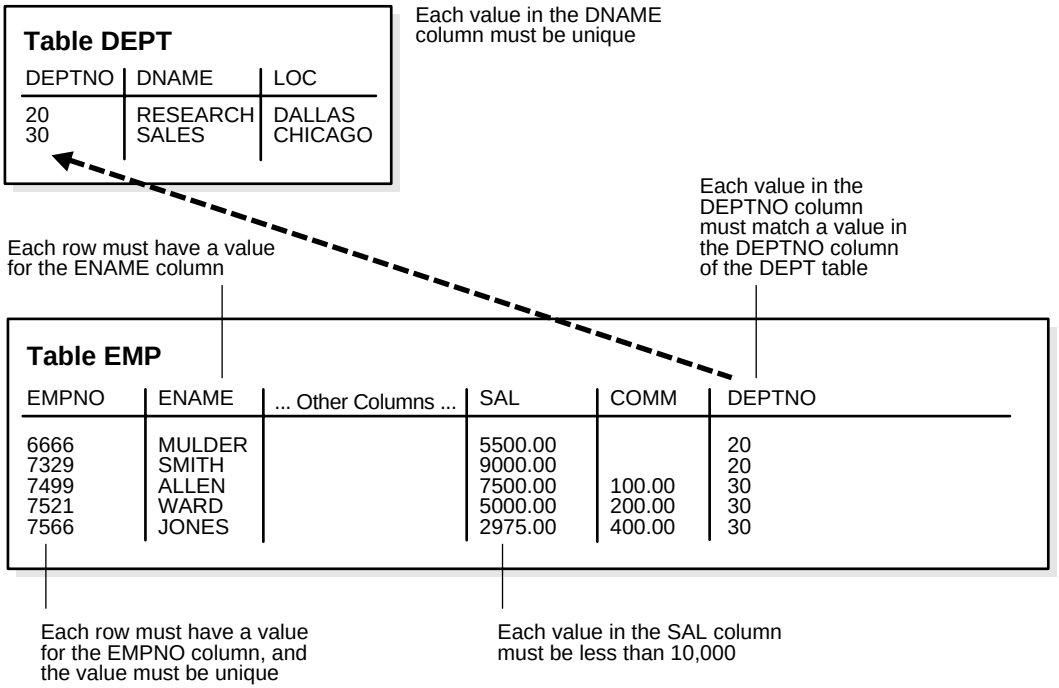
Data Integrity

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

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 25–1.

Figure 25–1 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.

Nulls

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

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

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 currently 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 nondeclarative approach using database triggers (stored database procedures automatically invoked on insert, update, or delete operations).

> **See Also:** Chapter 19, "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 19, "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 do not come without 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:

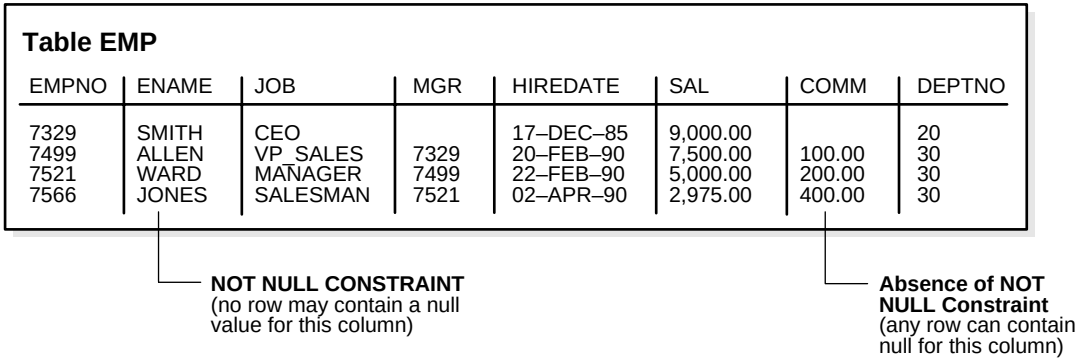
- NOT NULL Integrity Constraints
- UNIQUE Key Integrity Constraints
- PRIMARY KEY Integrity Constraints
- Referential Integrity Constraints
- CHECK Integrity Constraints

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 25–2 illustrates a NOT NULL integrity constraint.

Figure 25–2 NOT NULL Integrity Constraints

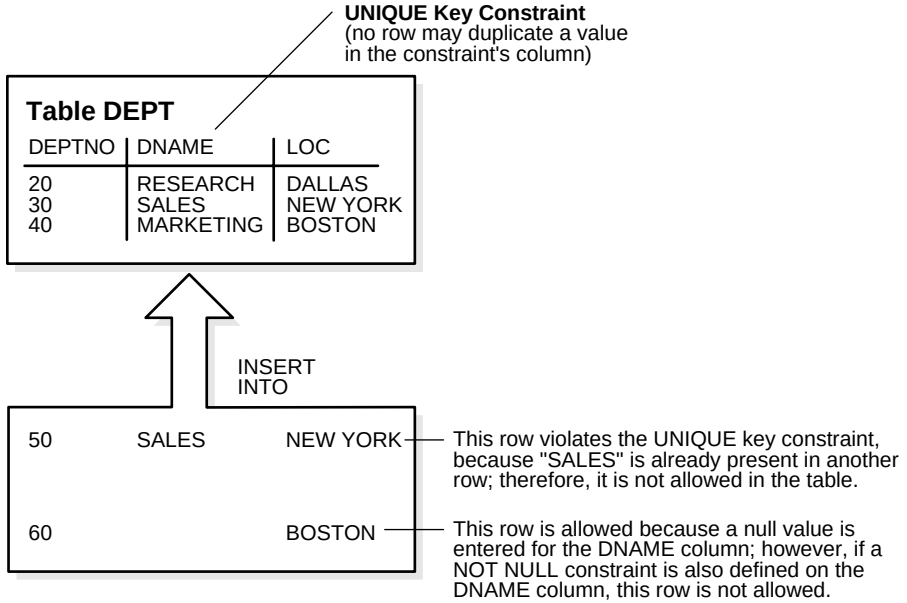


UNIQUE Key Integrity Constraints

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 25–3 a UNIQUE key constraint is defined on the DNAME column of the DEPT table to disallow rows with duplicate department names.

Figure 25–3 A UNIQUE Key Constraint

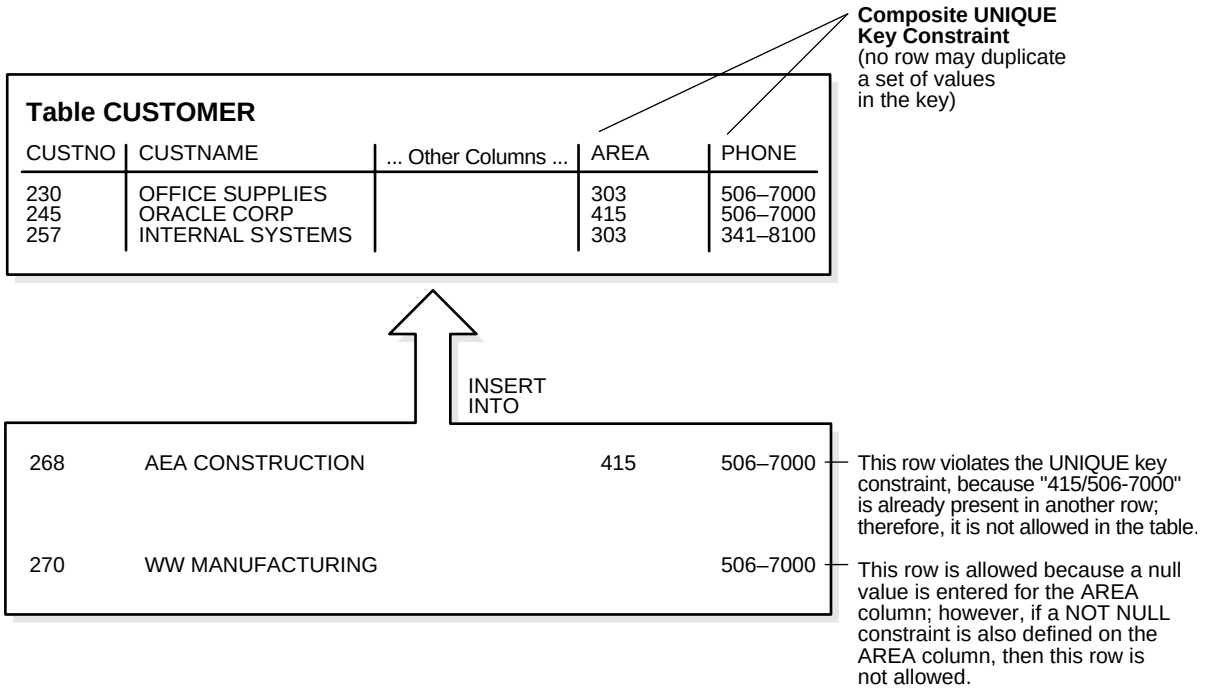


Unique Keys

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 25–4 the CUSTOMER table has a UNIQUE key constraint defined on the composite unique key: the AREA and PHONE columns.

Figure 25–4 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 25–4, 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, and the total size in bytes of a key value cannot 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.

Combining UNIQUE Key and NOT NULL Integrity Constraints

In Figure 25–3 and Figure 25–4, 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.

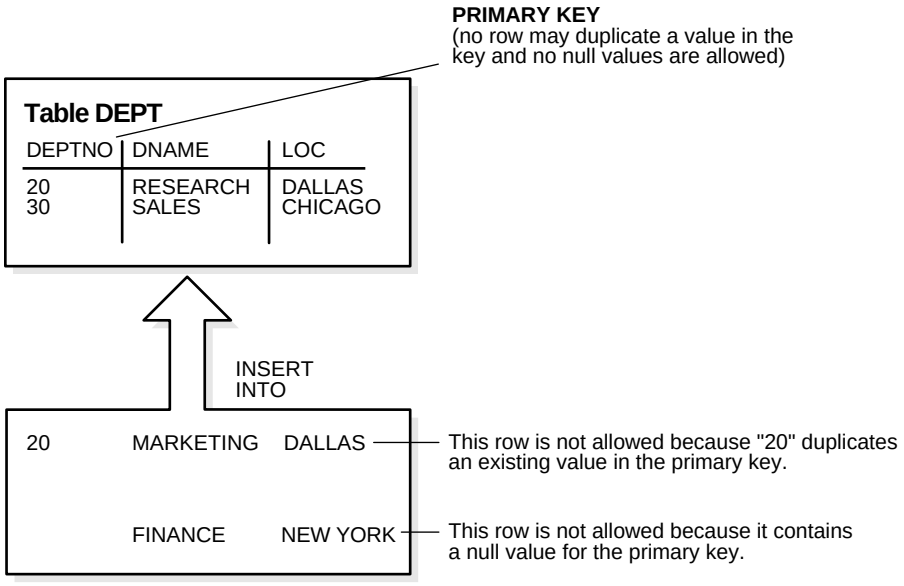
Primary Keys

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 25–5 illustrates a PRIMARY KEY constraint in the DEPT table and examples of rows that violate the constraint.

Figure 25–5 A Primary Key Constraint



PRIMARY KEY Constraints and Indexes

Oracle enforces all PRIMARY KEY constraints using indexes. In Figure 25–5, 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:

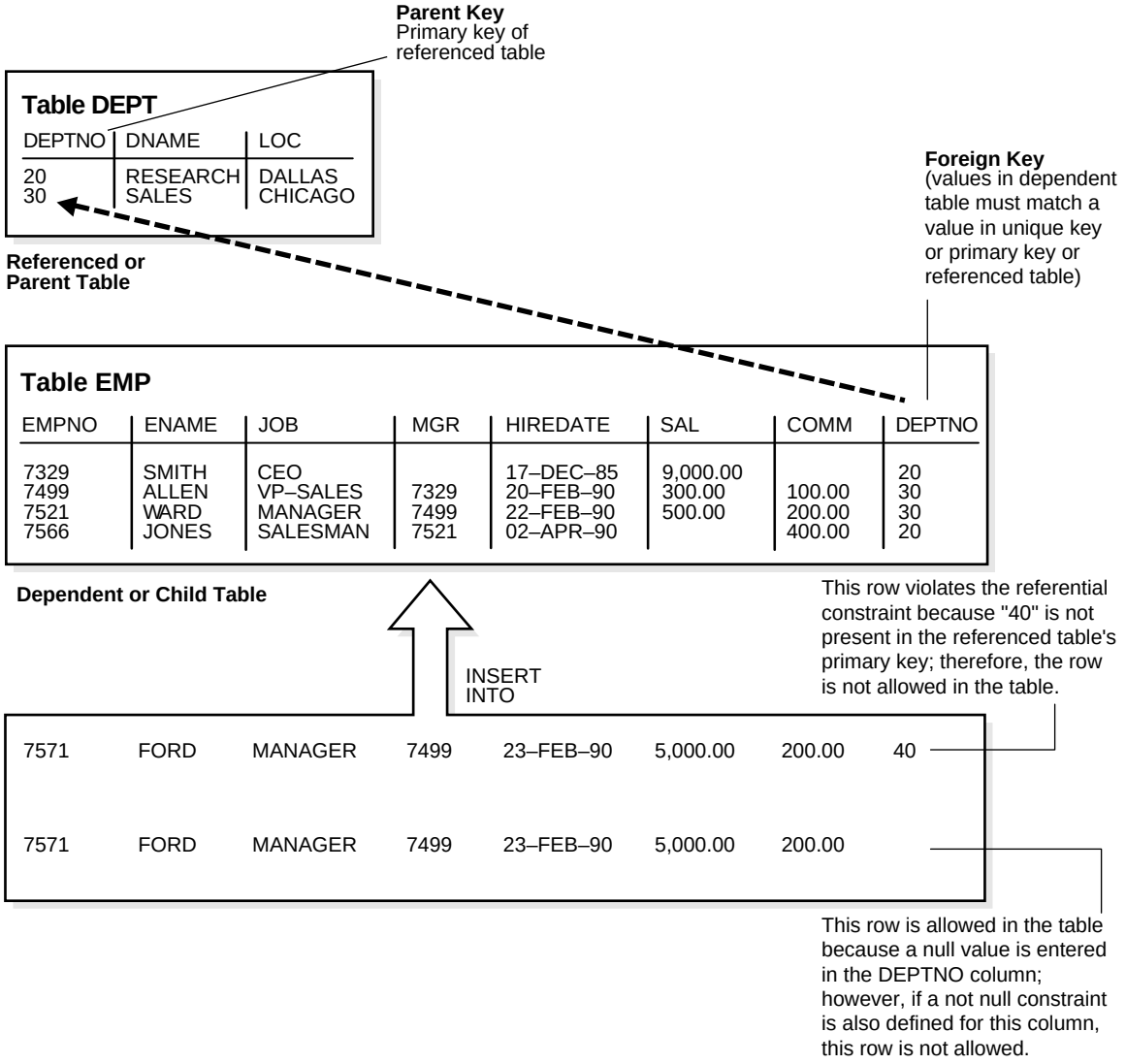
foreign key	The column or set of columns included in the definition of the referential integrity constraint that reference a referenced key.
referenced key	The unique key or primary key of the same or different table that is referenced by a foreign key.
dependent or child table	The table that includes 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 25–6 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 25–6 Referential Integrity Constraints

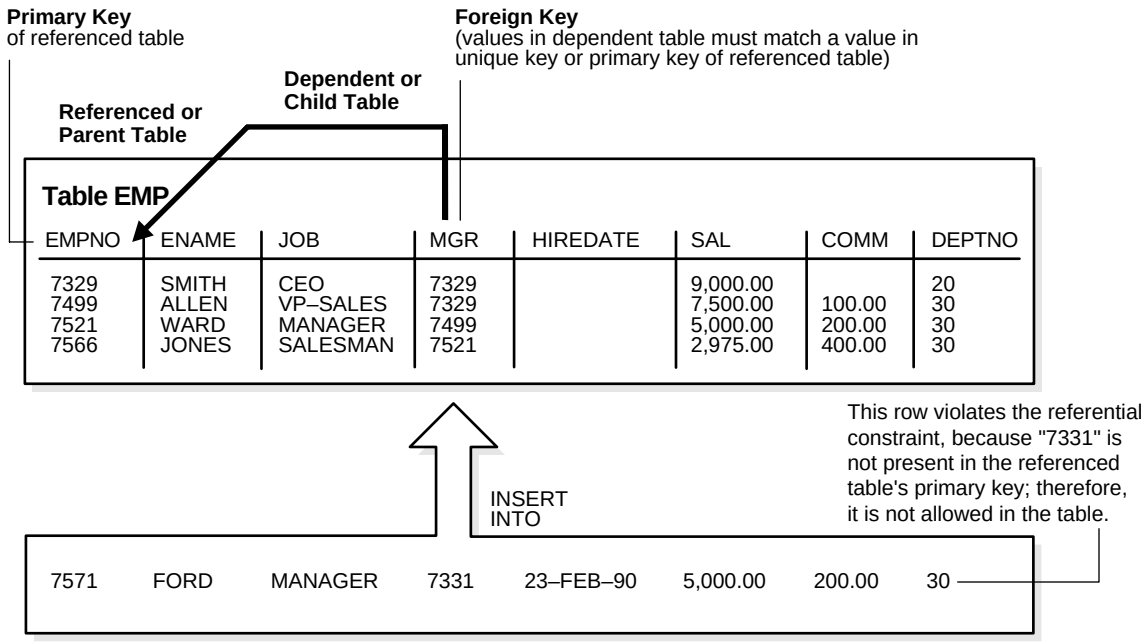


Self-Referential Integrity Constraints

Another type of referential integrity constraint, shown in Figure 25–7, is called a self-referential integrity constraint. This type of foreign key references a parent key in the same table.

In the example in Figure 25–7, 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 25–7 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. Several interpretations of this basic rule of the relational model are possible when composite (multicolumn) foreign keys are involved.

The ANSI/ISO SQL92 (entry-level) standard permits a composite foreign key to contain any value in its non-null columns if any other column is null, even if those non-null values are not found in the referenced key. By using other constraints such as NOT NULL and CHECK constraints, you can alter the treatment of partially null foreign keys from this default treatment.

A composite foreign key can be all null, all non-null, or partially null. The following terms define three alternative matching rules for composite foreign keys:

match full	Partially null foreign keys are not permitted. Either all components of the foreign key must be null, or the combination of values contained in the foreign key must appear as the primary or unique key value of a single row of the referenced table.
match partial	Partially null composite foreign keys are permitted. Either all components of the foreign key must be null, or the combination of non-null values contained in the foreign key must appear in the corresponding portion of the primary or unique key value of a single row in the referenced table.
match none	Partially null composite foreign keys are permitted. 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 19, "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 25–1 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 25–1 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 should 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.

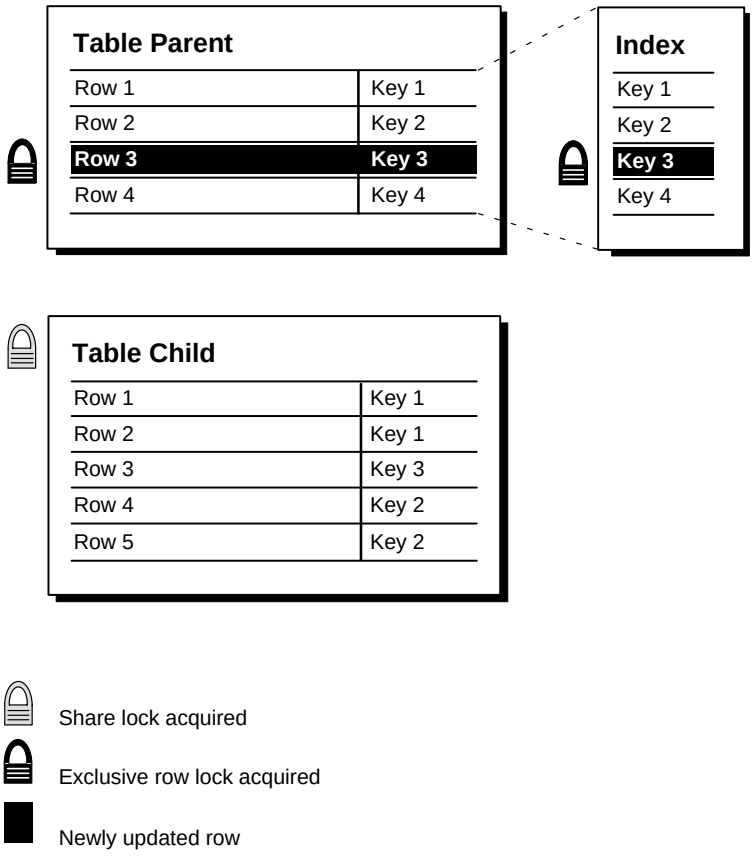
No Index on the Foreign Key Figure 25–8 illustrates the locking mechanisms used by Oracle when no index is defined on the foreign key and when rows are being updated or deleted in the parent table. Inserts into the parent table do not require any locks on the child table.

Notice that a share lock of the entire child table is required until the transaction containing the DELETE statement for the parent table is committed. If the foreign key specifies ON DELETE CASCADE, then the DELETE statement results in a table-level share-subexclusive lock on the child table. A share lock of the entire child table is also required for an UPDATE statement on the parent table that affects any columns referenced by the child table. Share locks allow reading only; therefore, no INSERT, UPDATE, or DELETE statements can be issued on the child table until the transaction containing the UPDATE or DELETE is committed. Queries are allowed on the child table.

This situation is tolerable if updates and deletes can be avoided on the parent.

INSERT, UPDATE, and DELETE statements on the child table do not acquire any locks on the parent table, although INSERT and UPDATE statements will wait for a row-lock on the index of the parent table to clear.

Figure 25–8 Locking Mechanisms When No Index Is Defined on the Foreign Key

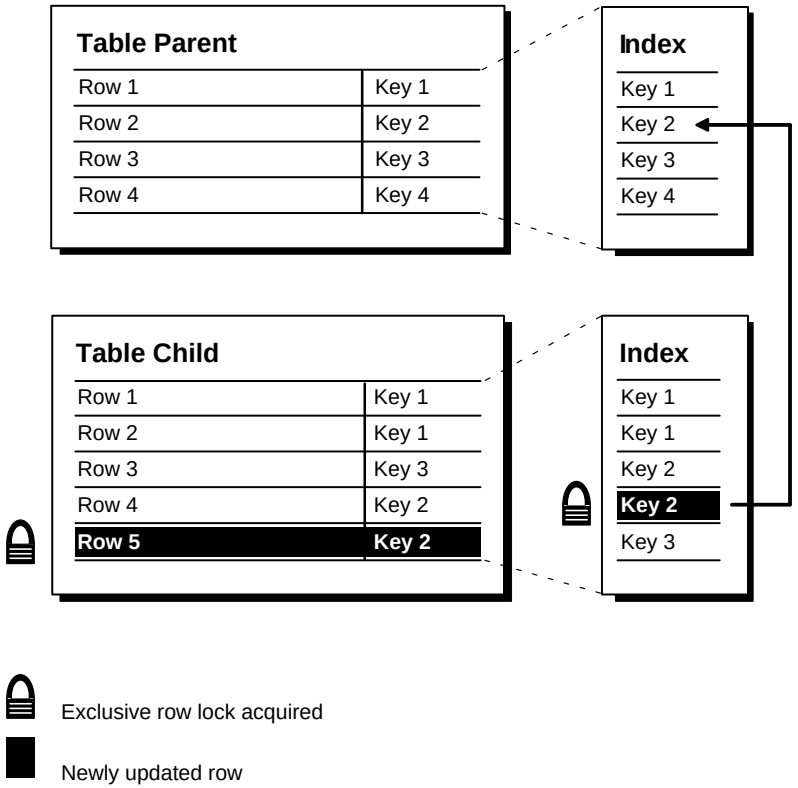


Index on the Foreign Key Figure 25–9 illustrates the locking mechanisms used by Oracle when an index is defined on the foreign key, and new rows are inserted, updated or deleted in the child table.

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 25–9 Locking Mechanisms When Index Is Defined on the Foreign Key



If the child table specifies ON DELETE CASCADE, then deletes from the parent table may 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 or sophisticated 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 NLS parameters as arguments (such as TO_CHAR, TO_DATE, and TO_NUMBER), Oracle uses the database's NLS settings by default. You can override the defaults by specifying NLS parameters explicitly in such functions within the CHECK constraint definition.

> **See Also:** *Oracle8i National Language Support Guide* for more information on NLS 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 25–7 on page 25-15.
- 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.

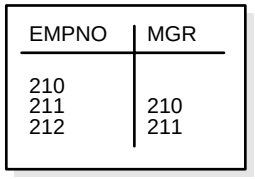
  This case also shows that constraint checking is deferred until the complete execution of the statement; all rows are inserted first, then all rows are checked for constraint violations. You can also defer the checking of constraints until the end of the *transaction*.

Consider the same self-referential integrity constraint in the following 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.

The table currently exists as illustrated in Figure 25–10.

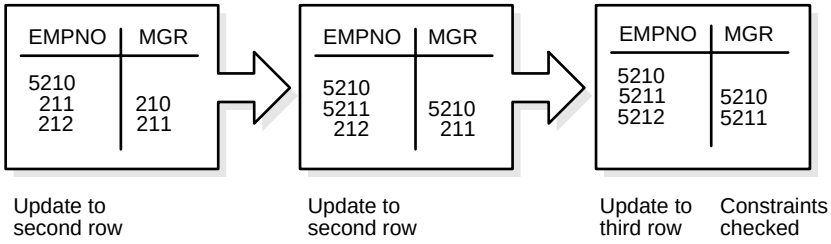
Figure 25–10 The EMP Table Before Updates




```
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 25–11 shows that Oracle performs the actions of the entire SQL statement before any constraints are checked.

Figure 25–11 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 25-24

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:

- *Oracle8i SQL Reference* for information about constraint attributes and their default values
- "Constraint States" on page 25-26
- "Modifying Constraint States" on page 25-27

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 signalled; at that point, a COMMIT would cause 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 snapshots, 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:** *Oracle8i Administrator's Guide* for more information about how to use the ENABLE, DISABLE, VALIDATE, and NOVALIDATE CONSTRAINT clauses

Modifying Constraint States

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:** *Oracle8i SQL Reference* for information about these constraint states

26

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
- 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 27, "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 above.

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 27, "Privileges, Roles, and Security Policies"
- *Oracle Advanced Security Administrator's Guide* for more information about enterprise users
- *Oracle8i 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 via 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:

- Chapter 30, "Distributed Database Concepts"
- Your Oracle operating system-specific documentation for more information about authenticating via your operating system

Authentication by the Network

Oracle supports several methods of authentication by the network, as described in the following sections.

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 Oracle8*i* Enterprise Edition with the Oracle Advanced Security option.

See Also:

- *Oracle8i Distributed Database Systems* 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:

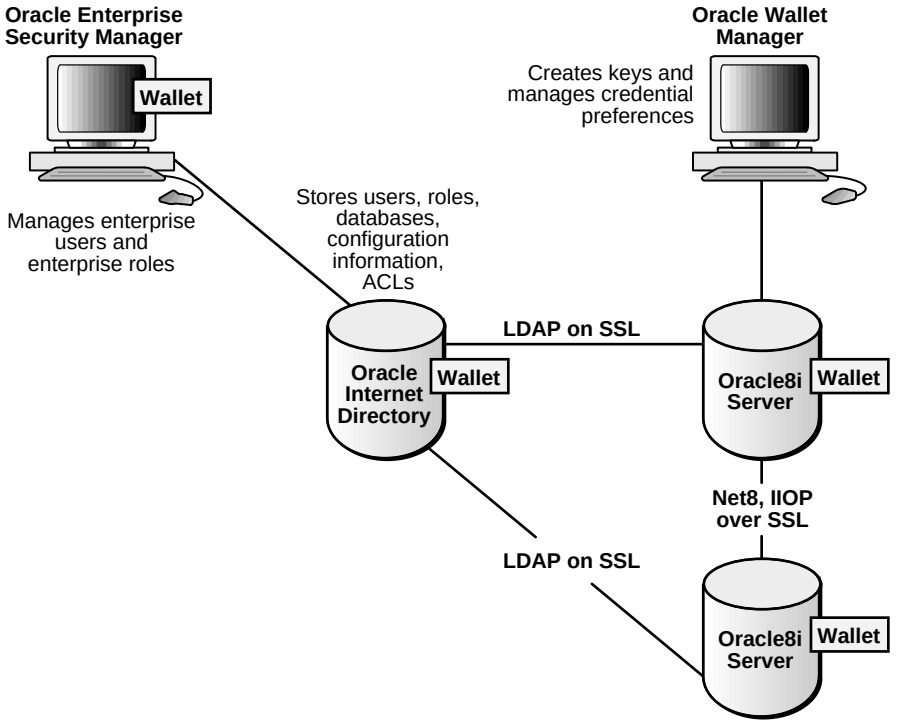
- Authentication and secure session key management using Secure Sockets Layer (SSL).
- Oracle Call Interface (OCI) and PL/SQL functions to sign user-specified data using a private key and certificate, and verify the signature on data using a certificate and trustpoint.
- *Oracle wallets*, which are data 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 protects user keys and manages X.509v3 certificates on Oracle clients and servers.
- X.509v3 certificates that you obtain from a certificate authority outside of Oracle. 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 Oracle8*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 simplifies user sign-on. The wallet must be configured with Oracle Wallet Manager first.

Oracle's public key infrastructure is illustrated in Figure 26–1.

Figure 26–1 Oracle Public Key Infrastructure



Note: To use public key infrastructure-based authentication with Oracle, you need Oracle8*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. To use remote authentication of users through RADIUS with Oracle, you need Oracle8*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: *Oracle8i Distributed Database Systems* 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 26-20

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

Multi-Tier Authentication and Authorization

In a multi-tier 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* via 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 multi-tier 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 multi-tier environment is based on trust regions, including the following:

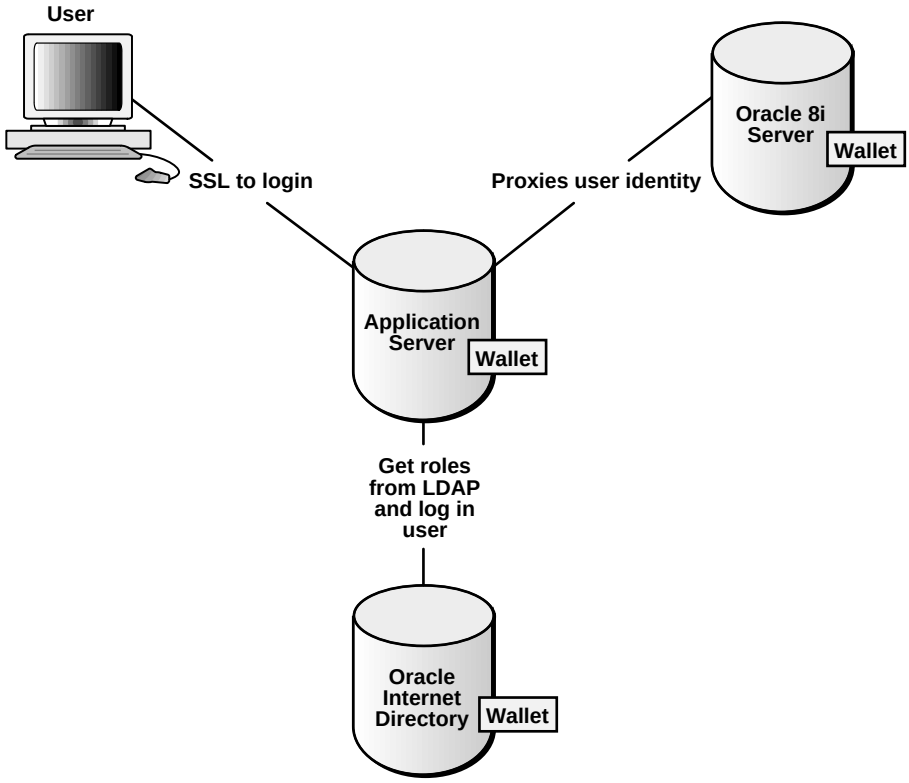
- The client provides proof of authentication to the application server, typically using a password or an X.509 certificate.
- The application server verifies the client authentication and then authenticates itself to the database server.
- The database server checks the application server authentication, verifies that the client exists, and verifies that the application server has the privilege to connect for this client.

Application servers may 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 26–2 shows an example of multi-tier authentication.

Figure 26–2 Multi-Tier 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 Multi-Tier Environment

Multi-tier 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 Multi-Tier Environment

Privileges in a multi-tier environment are limited to what is necessary to perform the requested operation.

Client Privileges Client privileges are as limited as possible in a multi-tier environment. Operations are performed on behalf of the client by the application server.

Application Server Privileges Application server privileges in a multi-tier environment are limited so that the application server cannot perform unwanted or unneeded operations while performing a client operation.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about multi-tier 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 may 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.

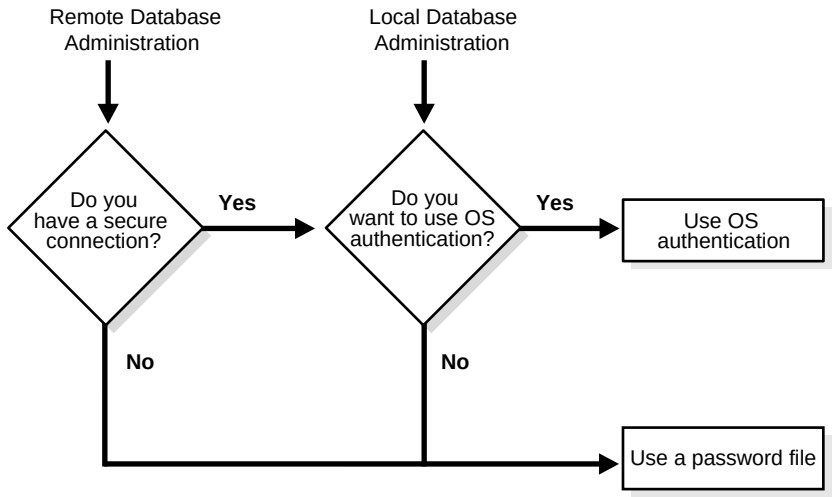
Authentication of Database Administrators

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 26–3 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 26–3 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
- *Oracle8i 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
- Temporary Tablespace
- Tablespace Access and Quotas

Default Tablespace

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; you can change it after the user has been created.

Temporary Tablespace

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, provided that 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 may 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 4, "Data Blocks, Extents, and Segments"
- Chapter 27, "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:** *Oracle8i 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 per 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 per call and per 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 per 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 per 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 multi-threaded 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: *Oracle8i 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.

Determining 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 the LOGICAL_READS_PER_SESSION and LOGICAL_READS_PER_CALL limits should be set 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 28, "Auditing"

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. (To upgrade an Oracle license, you must contact your Oracle representative.)

Note: When Oracle is embedded in an Oracle application (such as Oracle Office), 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: *Oracle8i 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. Once the warning limit is reached, Oracle allows additional sessions to connect (up to the maximum limit described above), but 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 the Oracle Parallel Server, 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 via database links. Once 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 an Oracle Parallel Server, all instances connected to the same database should have the same named user limit.

See Also:

- *Oracle8i Parallel Server Concepts*
- *Oracle8i Parallel Server Administration, Deployment, and Performance*

27

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

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:** *Oracle8i 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.

Granting and Revoking 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:** Usually, you should grant system privileges only to administrative personnel and application developers, because 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.

Granting and Revoking 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 SQL statements GRANT and REVOKE, respectively, or the Add Privilege to Role / User dialog box and Revoke Privilege from Role / User dialog box of Oracle Enterprise Manger.

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: *Oracle8i SQL Reference*

Table Security Topics

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. You should 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: *Oracle8i 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 25, "Data Integrity" for more information about primary keys, unique keys, and integrity constraints

View Security Topics

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:** *Oracle8i SQL Reference*

Increasing 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 Topics

The only *schema object privilege* for procedures, including standalone procedures and functions as well as packages, is EXECUTE. You should 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 27-20
- "Dependency Tracking for Stored Procedures" on page 17-11

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 19, "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 since 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:

- "Resolution of External References" on page 17-8
- "Fine-Grained Access Control" on page 27-23
- *Oracle8i Supplied PL/SQL Packages 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 via 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.

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.

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 Topics

This section describes privileges for types, methods, and objects.

System Privileges

Oracle8*i* defines system privileges shown in Table 27–1 for named types (object types, VARRAYs, and nested tables) :

Table 27–1 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 the above 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 Topics" on page 27-7

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 via 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 Topics" on page 27-5 for the requirements for creating a table

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 will 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 will fail 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. Oracle8*i* defines the privileges shown in Table 27–2 for object tables:

Table 27–2 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 may 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. Since a table contains data which relies on the type definition for access, any change to the type will cause all stored data to become inaccessible. Changes which can cause this effect are when necessary privileges required by the type are revoked or if the type or dependent types are dropped. If either of these actions occur, then table because invalid and cannot be accessed.

A table which is invalid because of missing privileges, may automatically become valid and accessible if the required privileges are re-granted. A table which 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 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 will always succeed and if there are depended upon tables, they will be invalidated.

> **See Also:** *Oracle8i Reference* for details about using the REVOKE, DROP TYPE, and FORCE clauses

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.

See Also:

- "Data Definition Language Statements and Roles" on page 27-21 for information about restrictions for procedures
- *Oracle8i Application Developer's Guide - Fundamentals* for instructions for enabling roles from an application

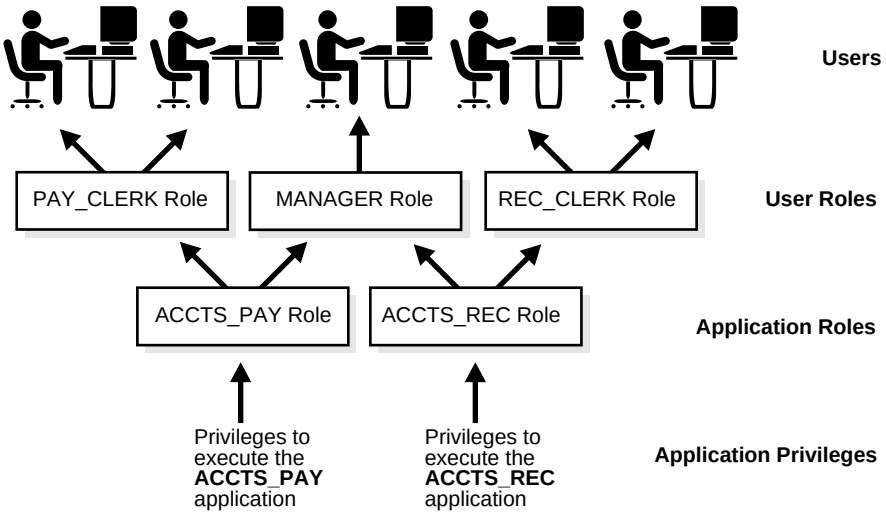
Common Uses for Roles

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 27–1 and the sections that follow describe the two uses of roles.

Figure 27–1 Common Uses for Roles



Application Roles

You grant an application role all privileges necessary to run a given database application. Then, you grant the 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 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.

Granting and Revoking 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:** *Oracle8i 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:** *Oracle8i Distributed Database Systems* for information about global roles

Naming Roles

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: *Oracle8i 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:

- "Dependency Tracking for Stored Procedures" on page 17-11 for an explanation of invoker and definer rights
- "Dynamic SQL in PL/SQL" on page 15-20

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.

  Examples:

  - 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.

  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:

Example: 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:** *Oracle8i Distributed Database Systems*

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.

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:

- "Packages" on page 17-12 for more information about using PL/SQL packages
- *Oracle8i Supplied PL/SQL Packages 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:** *Oracle8i 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" while 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 wish 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*
- *Oracle8i Supplied PL/SQL Packages Reference*

28

Auditing

This chapter discusses the auditing feature of Oracle. It includes:

- Introduction to Auditing
- Statement Auditing
- Privilege Auditing
- Schema Object Auditing
- Focusing Statement, Privilege, and Schema Object Auditing

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.

Auditing Features

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 per 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.

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 per 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 AUD$ 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, provided that 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; unsuccessful operations return the Oracle error code describing why the operation was unsuccessful.

See Also:

- *Oracle8i Administrator's Guide* for instructions for creating and using predefined views
- *Oracle8i Error Messages* for a list of completion codes

Auditing Mechanisms

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, 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:

- *Oracle8i Administrator's Guide* for instructions on enabling and disabling auditing
- Chapter 15, "SQL and PL/SQL" 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 to the database 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.

Auditing 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: Chapter 30, "Distributed Database Concepts"

Auditing 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.

Note: See your platform-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/SNAPSHOT statements, regardless of the table, view, or snapshot)

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, since 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 only 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:** *Oracle8i 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 above example, if auditing of SELECT statements is turned off for the EMP table, use of the EMP_DEPT view would no longer generate an audit record for the EMP table.

Focusing 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)

Auditing Successful and Unsuccessful Statement Executions

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

Auditing BY SESSION versus BY ACCESS

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: *Oracle8i 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, per 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.

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.

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 will contain 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.

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.

Auditing 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.

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:** *Oracle8i SQL Reference* for more information about auditing by user

29

Database Recovery

This chapter introduces the structures that are used during database recovery and describes the Recovery Manager utility, which simplifies backup and recovery operations. The topics in this chapter include:

- Introduction to Database Recovery
- Structures Used for Database Recovery
- Rolling Forward and Rolling Back
- Improving Recovery Performance
- Recovery Manager
- Database Archiving Modes
- Control Files
- Database Backups
- Survivability

See Also:

- *Oracle8i Recovery Manager User's Guide and Reference* for the procedures to create and maintain backup and recovery structures
- *Oracle8i Standby Database Concepts and Administration* for information about standby databases

Introduction to Database Recovery

A major responsibility of the database administrator is to prepare for the possibility of hardware, software, network, process, or system failure. If such a failure affects the operation of a database system, you must usually recover the database and return to normal operation as quickly as possible. Recovery should protect the database and associated users from unnecessary problems and avoid or reduce the possibility of having to duplicate work manually.

Recovery processes vary depending on the type of failure that occurred, the structures affected, and the type of recovery that you perform. If no files are lost or damaged, recovery may amount to no more than restarting an instance. If data has been lost, recovery requires additional steps.

Note: The Recovery Manager is a utility that simplifies backup and recovery operations.

See Also:

- "Recovery Manager" on page 29-16
- *Oracle8i Recovery Manager User's Guide and Reference* for detailed information on Recovery Manager and a description of how to recover from loss of data

Errors and Failures

Several problems can halt the normal operation of an Oracle database or affect database I/O to disk. The following sections describe the most common types. For some of these problems, recovery is automatic and requires little or no action on the part of the database user or database administrator.

User Error

A database administrator can do little to prevent user errors such as accidentally dropping a table. Usually, user error can be reduced by increased training on database and application principles. Furthermore, by planning an effective recovery scheme ahead of time, the administrator can ease the work necessary to recover from many types of user errors.

Statement Failure

Statement failure occurs when there is a logical failure in the handling of a statement in an Oracle program. For example, assume all extents of a table (in other words, the number of extents specified in the MAXEXTENTS parameter of the CREATE TABLE statement) are allocated, and are completely filled with data; the table is absolutely full. A valid INSERT statement cannot insert a row because there is no space available. Therefore, if issued, the statement fails.

If a statement failure occurs, the Oracle software or operating system returns an error code or message. A statement failure usually requires no action or recovery steps; Oracle automatically corrects for statement failure by rolling back the effects (if any) of the statement and returning control to the application. The user can simply re-execute the statement after correcting the problem indicated by the error message.

Process Failure

A process failure is a failure in a user, server, or background process of a database instance such as an abnormal disconnect or process termination. When a process failure occurs, the failed subordinate process cannot continue work, although the other processes of the database instance can continue.

The Oracle background process PMON detects aborted Oracle processes. If the aborted process is a user or server process, PMON resolves the failure by rolling back the current transaction of the aborted process and releasing any resources that this process was using. Recovery of the failed user or server process is automatic. If the aborted process is a background process, the instance usually cannot continue to function correctly. Therefore, you must shut down and restart the instance.

Network Failure

When your system uses networks such as local area networks and phone lines to connect client workstations to database servers, or to connect several database servers to form a distributed database system, network failures such as aborted phone connections or network communication software failures can interrupt the normal operation of a database system. For example:

- A network failure might interrupt normal execution of a client application and cause a process failure to occur. In this case, the Oracle background process PMON detects and resolves the aborted server process for the disconnected user process, as described in the previous section.
- A network failure might interrupt the two-phase commit of a distributed transaction. After the network problem is corrected, the Oracle background

process RECO of each involved database server automatically resolves any distributed transactions not yet resolved at all nodes of the distributed database system.

See Also: Chapter 30, "Distributed Database Concepts"

Database Instance Failure

Database instance failure occurs when a problem arises that prevents an Oracle database instance from continuing to work. An instance failure can result from a hardware problem, such as a power outage, or a software problem, such as an operating system crash. Instance failure also results when you issue a SHUTDOWN ABORT or STARTUP FORCE statement.

Recovery from Instance Failure Crash or instance recovery recovers a database to its transaction-consistent state just before instance failure. *Crash recovery* recovers a database in a single-instance configuration and *instance recovery* recovers a database in an Oracle Parallel Server configuration. If all instances of an Oracle Parallel Server database crash, then Oracle performs crash recovery.

Recovery from instance failure is automatic, requiring no DBA intervention. For example, when using the Oracle Parallel Server, another instance performs instance recovery for the failed instance. In single-instance configurations, Oracle performs crash recovery for a database when the database is restarted, that is, mounted and opened to a new instance. The transition from a mounted state to an open state automatically triggers crash recovery, if necessary.

Crash or instance recovery consists of the following steps:

1. Rolling forward to recover data that has not been recorded in the datafiles, yet has been recorded in the online redo log, including the contents of rollback segments. This is called *cache recovery*.
2. Opening the database. Instead of waiting for all transactions to be rolled back before making the database available, Oracle allows the database to be opened as soon as cache recovery is complete. Any data that is not locked by unrecovered transactions is immediately available.
3. Marking all transactions system-wide that were active at the time of failure as DEAD and marking the rollback segments containing these transactions as PARTLY AVAILABLE.
4. Rolling back dead transactions as part of SMON recovery. This is called *transaction recovery*.

5. Resolving any pending distributed transactions undergoing a two-phase commit at the time of the instance failure.
6. As new transactions encounter rows locked by dead transactions, they can automatically roll back the dead transaction to release the locks. If you are using Fast-Start Recovery, just the data block is immediately rolled back, as opposed to the entire transaction.

See Also:

- *Oracle8i Parallel Server Setup and Configuration Guide* for a discussion of instance recovery
- *Oracle8i Designing and Tuning for Performance* for a discussion of instance recovery tuning

Media (Disk) Failure

An error can arise when trying to write or read a file that is required to operate an Oracle database. This occurrence is called *media failure* because there is a physical problem reading or writing to files on the storage medium.

A common example of media failure is a disk head crash, which causes the loss of all files on a disk drive. All files associated with a database are vulnerable to a disk crash, including datafiles, online redo log files, and control files.

The appropriate recovery from a media failure depends on the files affected.

See Also:

- *Oracle8i Backup and Recovery Guide* for a discussion of recovery methods
- *Oracle8i Recovery Manager User's Guide and Reference*

How Media Failures Affect Database Operation Media failures can affect one or all types of files necessary for the operation of an Oracle database, including datafiles, online redo log files, and control files.

Database operation after a media failure of online redo log files or control files depends on whether the online redo log or control file is *multiplexed*, as recommended. A multiplexed online redo log or control file means that a second copy of the file is maintained. If a media failure damages a single disk, and you have a multiplexed online redo log, the database can usually continue to operate without significant interruption. Damage to a non-multiplexed online redo log causes database operation to halt and may cause permanent loss of data. Damage to

any control file, whether it is multiplexed or non-multiplexed, halts database operation once Oracle attempts to read or write the damaged control file, which happens frequently, for example at every checkpoint and log switch.

Media failures that affect datafiles can be divided into two categories: read errors and write errors. In a read error, Oracle discovers it cannot read a datafile and an operating system error is returned to the application, along with an Oracle error indicating that the file cannot be found, cannot be opened, or cannot be read. Oracle continues to run, but the error is returned each time an unsuccessful read occurs. At the next checkpoint, a write error will occur when Oracle attempts to write the file header as part of the standard checkpoint process.

If Oracle discovers that it cannot write to a datafile and Oracle is in ARCHIVELOG mode, then Oracle returns an error in the DBW*n* trace file and takes the datafile offline automatically. Only the datafile that cannot be written to is taken offline; the tablespace containing that file remains online.

If the datafile that cannot be written to is in the SYSTEM tablespace, then the file is not taken offline. Instead, an error is returned and Oracle shuts down the instance. The reason for this exception is that all files in the SYSTEM tablespace must be online in order for Oracle to operate properly. For the same reason, the datafiles of a tablespace containing active rollback segments must remain online.

If Oracle discovers that it cannot write to a datafile, and Oracle is not archiving the filled online redo log files, then the DBW*n* background process fails and the current instance fails. If the problem is temporary (for example, the disk controller was powered off), then crash or instance recovery usually can be performed using the online redo log files, in which case the instance can be restarted. However, if a datafile is permanently damaged and archiving is not used, then the entire database must be restored using the most recent consistent backup.

Recovery of Read-Only Tablespaces Recovery is not needed on read-only datafiles during crash or instance recovery. Recovery during startup verifies that each online read-only file does not need any media recovery. That is, the file was not restored from a backup taken before it was made read-only. If you restore a read-only tablespace from a backup taken before the tablespace was made read-only, then you cannot access the tablespace until you complete media recovery.

Structures Used for Database Recovery

Several structures of an Oracle database safeguard data against possible failures. This section introduces each of these structures and its role in database recovery.

Database Backups

A database backup consists of backups of the physical files (all datafiles and a control file) that constitute an Oracle database. To begin media recovery after a media failure, Oracle uses file backups to restore damaged datafiles or control files. Replacing a current, possibly damaged, copy of a datafile, tablespace, or database with a backup copy is called *restoring* that portion of the database.

Oracle offers several options in performing database backups, including:

- Recovery Manager
- Operating system utilities
- Export utility
- Enterprise Backup Utility

See Also: *Oracle8i Backup and Recovery Guide*

The Redo Log

The redo log, present for every Oracle database, records all changes made in an Oracle database. The redo log of a database consists of at least two redo log files that are separate from the datafiles (which actually store a database's data). As part of database recovery from an instance or media failure, Oracle applies the appropriate changes in the database's redo log to the datafiles, which updates database data to the instant that the failure occurred.

A database's redo log can consist of two parts: the online redo log and the archived redo log.

The Online Redo Log

Every Oracle database has an associated online redo log. The Oracle background process LGWR uses the online redo log to immediately record all changes made through the associated instance. Each redo record contains both the old and the new values. Oracle also records the old value to a rollback block. The online redo log consists of two or more pre-allocated files that are reused in a circular fashion to record ongoing database changes.

The Archived Redo Log

Optionally, you can configure an Oracle database to archive files of the online redo log once they fill. The online redo log files that are archived are uniquely identified by their log sequence numbers and make up the archived redo log. By archiving

filled online redo log files, older redo log information is preserved for operations such as media recovery, while the pre-allocated online redo log files continue to be reused to store the most current database changes.

Datafiles that were restored from backup, or were not closed by a clean database shutdown, may not be completely up to date. These datafiles must be updated by applying the changes in the archived and / or online redo logs. This process is called *recovery*.

See Also: "Database Archiving Modes" on page 29-18

Rollback Segments

Rollback segments are used for a number of functions in the operation of an Oracle database. In general, the rollback segments of a database store the old values of data changed by ongoing transactions for uncommitted transactions.

Among other things, the information in a rollback segment is used during database recovery to undo any uncommitted changes applied from the redo log to the datafiles. Therefore, if database recovery is necessary, then the data is in a consistent state after the rollback segments are used to remove all uncommitted data from the datafiles.

Control Files

In general, the control files of a database store the status of the physical structure of the database. Status information in the control file such as the current online redo log file and the names of the datafiles guides Oracle during instance or media recovery.

See Also: "Control Files" on page 29-22

Rolling Forward and Rolling Back

Database buffers in the buffer cache in the SGA are written to disk only when necessary, using a least-recently-used algorithm. Because of the way that the DBW*n* process uses this algorithm to write database buffers to datafiles, datafiles might contain some data blocks modified by uncommitted transactions and some data blocks missing changes from committed transactions.

Two potential problems can result if an instance failure occurs:

- Data blocks modified by a transaction might not be written to the datafiles at commit time and might only appear in the redo log. Therefore, the redo log contains changes that must be reapplied to the database during recovery.
- After the roll forward phase, the datafiles may contain changes that had not been committed at the time of the failure. These uncommitted changes must be rolled back to ensure transactional consistency. These changes were either saved to the datafiles before the failure, or introduced during the roll forward phase.

To solve this dilemma, two separate steps are generally used by Oracle for a successful recovery of a system failure: rolling forward with the redo log (cache recovery) and rolling back with the rollback segments (transaction recovery).

The Redo Log and Rolling Forward

The redo log is a set of operating system files that record all changes made to any database buffer, including data, index, and rollback segments, *whether the changes are committed or uncommitted.* Each redo entry is a group of change vectors describing a single atomic change to the database. The redo log protects changes made to database buffers in memory that have not been written to the datafiles.

The first step of recovery from an instance or disk failure is to *roll forward*, or reapply all of the changes recorded in the redo log to the datafiles. Because rollback data is also recorded in the redo log, rolling forward also regenerates the corresponding rollback segments. This is called *cache recovery.*

Rolling forward proceeds through as many redo log files as necessary to bring the database forward in time. Rolling forward usually includes online redo log files and may include archived redo log files.

After roll forward, the data blocks contain all committed changes. They may also contain uncommitted changes that were either saved to the datafiles before the failure, or were recorded in the redo log and introduced during roll forward.

Rollback Segments and Rolling Back

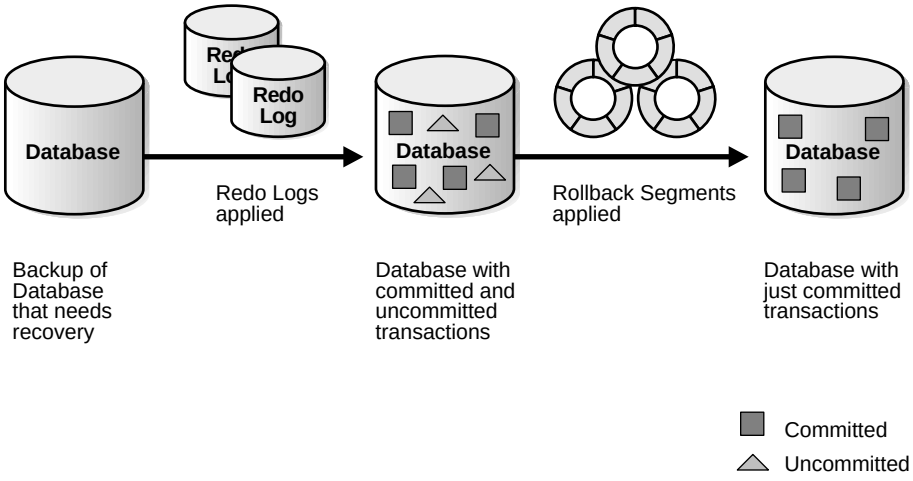
Rollback segments record database actions that should be undone during certain database operations. In database recovery, rollback segments undo the effects of uncommitted transactions previously applied by the rolling forward phase.

After the roll forward, any changes that were not committed must be undone. After redo log files have reapplied all changes made to the database, then the corresponding rollback segments are used. Rollback segments are used to identify and undo transactions that were never committed, yet were either saved to the

datafiles before the failure, or were applied to the database during the roll forward. This process is called *rolling back* or *transaction recovery*.

Figure 29–1 illustrates rolling forward and rolling back, the two steps necessary to recover from any type of system failure.

Figure 29–1 Basic Recovery Steps: Rolling Forward and Rolling Back



Oracle can roll back multiple transactions simultaneously as needed. All transactions system-wide that were active at the time of failure are marked as DEAD. Instead of waiting for SMON to roll back dead transactions, new transactions can recover blocking transactions themselves to get the row locks they need.

Improving Recovery Performance

When a database failure occurs, rapid recovery is very important in most situations. Oracle provides a number of methods to make recovery as quick as possible, including:

- Parallel Recovery
- Fast-Start Recovery
- Transparent Application Failover

Performing Recovery in Parallel

Recovery reapplies the changes generated by several concurrent processes, and therefore instance or media recovery can take longer than the time it took to initially generate the changes to a database. With serial recovery, a single process applies the changes in the redo log files sequentially. Using parallel recovery, several processes simultaneously apply changes from redo log files.

> **Note:** Oracle8*i* provides limited parallelism with Recovery Manager; the Oracle8*i* Enterprise Edition allows unlimited parallelism. See *Getting to Know Oracle8i* for more information about the features available in Oracle8*i* and Oracle8*i* Enterprise Edition.

Parallel recovery can be performed using these methods:

- Parallel recovery can be performed manually by spawning several Oracle Enterprise Manager sessions and issuing the RECOVER DATAFILE command on a different set of datafiles in each session. However, this method causes each Oracle Enterprise Manager session to read the entire redo log file.
- You can use the Recovery Manager's **restore** and **recover** statements to automatically parallelize all stages of recovery. Oracle uses one process to read the log files sequentially and dispatch redo information to several recovery processes, which apply the changes from the log files to the datafiles. The recovery processes are started automatically by Oracle, so there is no need to use more than one session to perform recovery. There are also some initialization parameters to set for automatic parallel recovery.
- You can use the SQL*Plus RECOVER statement to perform parallel recovery. Refer to the *SQL*Plus User's Guide and Reference* for details.
- You can use the SQL statement ALTER DATABASE RECOVER to perform parallel recovery, but this is not recommended.

> **See Also:** *Oracle8i Parallel Server Setup and Configuration Guide* for information about setting initialization parameters for automatic parallel recovery

Situations That Benefit from Parallel Recovery

In general, parallel recovery is most effective at reducing recovery time when several datafiles on several different disks are being recovered concurrently. Crash

recovery (recovery after instance failure) and media recovery of many datafiles on many different disk drives are good candidates for parallel recovery.

The performance improvement from parallel recovery is also dependent upon whether the operating system supports asynchronous I/O. If asynchronous I/O is not supported, parallel recovery can dramatically reduce recovery time. If asynchronous I/O is supported, the recovery time may only be slightly reduced by using parallel recovery.

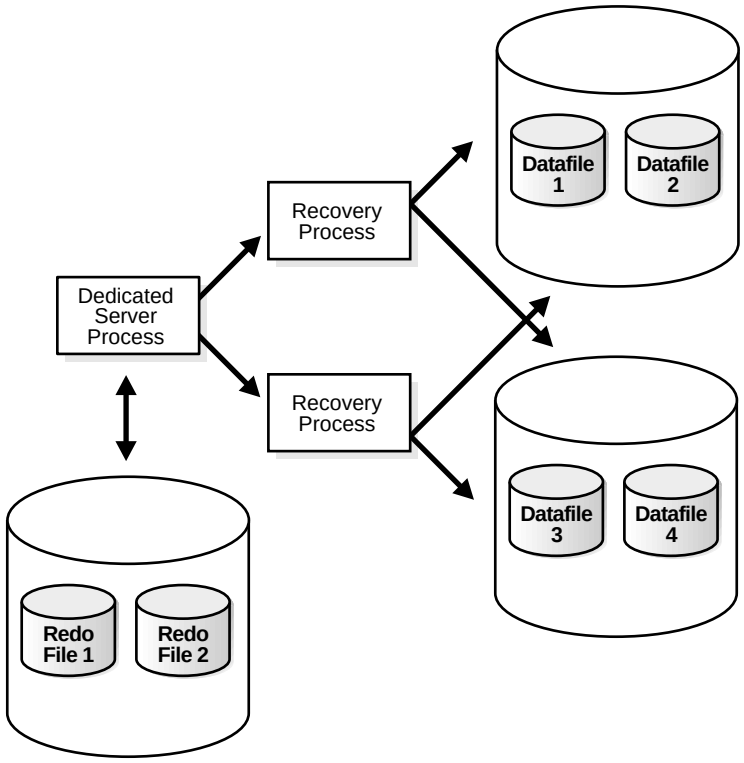
> **See Also:** Your operating system documentation to determine whether the system supports asynchronous I/O

Recovery Processes

In a typical parallel recovery situation, one process is responsible for reading and dispatching redo entries from the redo log files. This is the dedicated server process that begins the recovery session. The server process reading the redo log files enlists two or more recovery processes to apply the changes from the redo entries to the datafiles.

Figure 29–2 illustrates a typical parallel recovery session.

Figure 29–2 Typical Parallel Recovery Session



In most situations, one recovery session and one or two recovery processes per disk drive containing datafiles needing recovery is sufficient. Recovery is a disk-intensive activity as opposed to a CPU-intensive activity, and therefore the number of recovery processes needed is dependent entirely upon how many disk drives are involved in recovery. In general, a minimum of eight recovery processes is needed before parallel recovery can show improvement over a serial recovery.

Fast-Start Fault Recovery

Fast-start fault recovery is an architecture that reduces the time required for rolling forward and makes the recovery bounded and predictable. It also eliminates rollback time from recovery for transactions aborted due to system faults. Fast-start fault recovery includes:

- Fast-Start Checkpointing
- Fast-Start On-Demand Rollback
- Fast-Start Parallel Rollback

Fast-Start Checkpointing

Fast-start checkpointing records the position in the redo thread (log) from which crash or instance recovery would need to begin. This position is determined by the oldest dirty buffer in the buffer cache. Each DBW*n* process continually writes buffers to disk to advance the checkpoint position, with minimal or no overhead during normal processing. Fast-start checkpointing improves the performance of crash and instance recovery, but not media recovery.

You can influence recovery performance for situations where there are stringent limitations on the duration of crash or instance recovery. The time required for crash or instance recovery is roughly proportional to the number of data blocks that need to be read or written during the roll forward phase. You can specify a limit, or bound, on the number of data blocks that will need to be processed during roll forward. The Oracle server automatically adjusts the checkpoint write rate to meet the specified roll-forward bound while issuing the minimum number of writes.

You can set the dynamic initialization parameter FAST_START_IO_TARGET to limit the number of blocks that need to be read for crash or instance recovery. Smaller values of this parameter impose higher overhead during normal processing because more buffers have to be written. On the other hand, the smaller the value of this parameter, the better the recovery performance, since fewer blocks need to be recovered. The dynamic initialization parameters LOG_CHECKPOINT_INTERVAL and LOG_CHECKPOINT_TIMEOUT also influence Fast-Start Checkpointing.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for information about how to set the value of FAST_START_IO_TARGET

Fast-Start On-Demand Rollback

When a dead transaction holds a row lock on a row that another transaction needs, fast-start on-demand rollback immediately recovers only the data block under consideration, leaving the rest of the dead transaction to be recovered in the background. This improves the availability of the database for users accessing data that is locked by large dead transactions. If fast-start on-demand rollback is not enabled, the user would have to wait until the entire dead transaction was recovered before obtaining the row lock.

Fast-Start Parallel Rollback

Fast-start parallel rollback allows a set of transactions to be recovered in parallel using a group of server processes. This technique is used when SMON determines that the amount of work it takes to perform recovery in parallel is less than the time it takes to recovery serially.

Masking Failures with Transparent Application Failover

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 re-establish connections to the database, and work in progress may be lost. Oracle8*i* Transparent Application Failover (TAF) can mask many failures from users, preserving the state of their applications 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:** *Oracle8i Designing and Tuning for Performance* for more information about Transparent Application Failover

Recovery Manager

Recovery Manager is a utility that *manages* the processes of creating backups of all database files (datafiles, control files, and archived redo log files) and restoring or recovering files from backups.

> **See Also:** *Oracle8i Recovery Manager User's Guide and Reference* for a full description of Recovery Manager

Recovery Catalog

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 commands called *stored scripts*

The recovery catalog is maintained solely by Recovery Manager. The database server of the backed-up database never accesses the recovery catalog directly. Recovery Manager propagates information about backup datafile sets, archived redo logs, backup control files, and datafile copies into the recovery catalog for long-term retention.

When doing a restore, Recovery Manager extracts the appropriate information from the recovery catalog and passes it to the database server. The server performs various integrity checks on the input files specified for a restore. Incorrect behavior by Recovery Manager cannot corrupt the database.

The Recovery Catalog Database

The recovery catalog is stored in an Oracle database. It is the database administrator's responsibility to make such a database available to Recovery Manager. Taking backups of the recovery catalog is also the database administrator's responsibility. Because the recovery catalog is stored in an Oracle database, you can use Recovery Manager to back it up.

If the recovery catalog is destroyed and no backups are available, then it can be partially reconstructed from the current control file or control file backups.

Operation Without a Recovery Catalog

Use of a recovery catalog is not required, but is recommended. Because most information in the recovery catalog is also available from the control file, Recovery Manager supports an operational mode where it uses only the control file. This operational mode is appropriate for small databases where installation and administration of another database to serve as the recovery catalog would be burdensome.

Some Recovery Manager features are only available when a recovery catalog is used.

> **See Also:** *Oracle8i Recovery Manager User's Guide and Reference* for information about creating the recovery catalog, and about which Recovery Manager features require use of a recovery catalog

Parallelization

Recovery Manager can parallelize its operations, establishing multiple logon sessions and conducting multiple operations in parallel by using non-blocking UPI. Concurrent operations must operate on disjoint sets of datafiles.

> **Note:** The Oracle8*i* Enterprise Edition allows unlimited parallelism. Oracle8*i* can only allocate one Recovery Manager channel at a time, thus limiting the parallelism to one stream. See *Getting to Know Oracle8i* for more information about the features available with Oracle8*i* and Oracle8*i* Enterprise Edition.

Parallelization of the **backup**, **copy**, and **restore** commands is handled internally by Recovery Manager. You only need to specify:

- A list of one or more sequential I/O devices such as disk or tape drives
- The objects to be backed up, copied, or restored.

Recovery Manager executes commands serially, that is, it completes the previous command before starting the next command. Parallelism is exploited only within the context of a single command. Thus, if 10 datafile copies are desired, it is better to issue a single **copy** command that specifies all 10 copies rather than 10 separate **copy** commands.

Report Generation

The **report** and **list** commands provide information about backups and image copies. The output from these commands is written to the message log file.

The **report** command produces reports that can answer questions such as:

- What files need a backup
- What files haven't had a backup in a while
- What backup files can be deleted

You can use the **report need backup** and **report unrecoverable** commands on a regular basis to ensure that the necessary backups are available to perform recovery, and that the recovery can be performed within a reasonable length of time.

A datafile is considered *unrecoverable* if an unlogged operation has been performed against a schema object residing in the datafile.

(A datafile that does not have a backup is not considered unrecoverable. Such datafiles can be recovered through the use of the CREATE DATAFILE statement, provided that logs starting from when the file was created still exist.)

The **list** command queries the recovery catalog and produces a listing of its contents. You can use it to find out what backups or copies are available:

- Backups or copies of a specified list of datafiles
- Backups or copies of any datafile that is a member of a specified list of tablespaces
- Backups or copies of any archive logs with a specified name and / or within a specified range
- Incarnations of a specified database

Database Archiving Modes

A database can operate in two distinct modes: NOARCHIVELOG mode (media recovery disabled) or ARCHIVELOG mode (media recovery enabled).

NOARCHIVELOG Mode (Media Recovery Disabled)

If a database is used in NOARCHIVELOG mode, then the archiving of the online redo log is disabled. Information in the database's control file indicates that filled

groups are not required to be archived. Therefore, as soon as a filled group becomes inactive, the group is available for reuse by the LGWR process.

NOARCHIVELOG mode protects a database only from instance failure, not from disk failure. Only the most recent changes made to the database, stored in the groups of the online redo log, are available for crash recovery or instance recovery. This is sufficient to satisfy the needs of crash recovery and instance recovery, because Oracle will not overwrite an online redo log that might be needed until its changes have been safely recorded in the datafiles. However, it will not be possible to do media recovery.

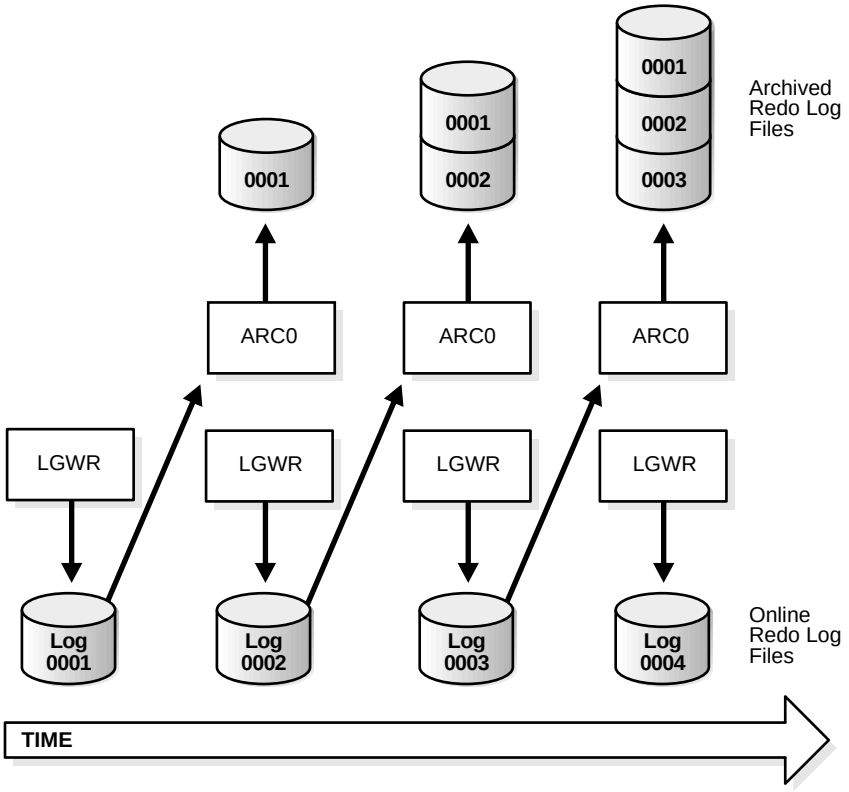
ARCHIVELOG Mode (Media Recovery Enabled)

If an Oracle database is operated in ARCHIVELOG mode, the archiving of the online redo log is enabled. Information in a database control file indicates that a group of filled online redo log files cannot be reused by LGWR until the group has been archived.

Figure 29–3 illustrates how the database's online redo log files are used in ARCHIVELOG mode and how the archived redo log is generated by the process archiving the filled groups (for example, ARC0 in this illustration).

ARCHIVELOG mode permits complete recovery from disk failure as well as instance failure, because all changes made to the database are permanently saved in an archived redo log.

Figure 29–3 Online Redo Log File Use in ARCHIVELOG Mode



Automatic Archiving and the ARC*n* (Archiver) Background Processes

An instance can be configured to have an additional background process, the archiver (ARC0), automatically archive groups of online redo log files after they become inactive. Automatic archiving frees the database administrator from having to keep track of, and archive, filled groups manually. For this convenience alone, automatic archiving is the choice of most database systems that run in ARCHIVELOG mode. For heavy workloads, such as bulk loading of data, multiple archiver processes (up to ARC9) can be configured by setting the initialization parameter LOG_ARCHIVE_MAX_PROCESSES.

If you request automatic archiving at instance startup by setting the LOG_ARCHIVE_START initialization parameter, Oracle starts the number of

ARC*n* processes specified by LOG_ARCHIVE_MAX_PROCESSES during instance startup. Otherwise, the ARC*n* processes are not started when the instance starts up.

However, the database administrator can interactively start or stop automatic archiving at any time. If automatic archiving was not specified to start at instance startup, and the administrator subsequently starts automatic archiving, Oracle creates the ARC*n* background process(es). ARC*n* then remains for the duration of the instance, even if automatic archiving is temporarily turned off and turned on again, although the number of ARC*n* processes can be changed dynamically by setting LOG_ARCHIVE_MAX_PROCESSES with the ALTER SYSTEM statement.

ARC*n* always archives groups in order, beginning with the lowest sequence number. ARC*n* automatically archives filled groups as they become inactive. A record of every automatic archival is written in the ARC*n* trace file by the ARC*n* process. Each entry shows the time the archive started and stopped.

If ARC*n* encounters an error when attempting to archive a log group (for example, due to an invalid or filled destination), ARC*n* continues trying to archive the group. An error is also written in the ARC*n* trace file and the ALERT file. If the problem is not resolved, eventually all online redo log groups become full, yet not archived, and the system halts because no group is available to LGWR. Therefore, if problems are detected, you should either resolve the problem so that ARC*n* can continue archiving (such as by changing the archive destination) or manually archive groups until the problem is resolved.

Manual Archiving

If a database is operating in ARCHIVELOG mode, the database administrator can manually archive the filled groups of inactive online redo log files, as necessary, whether or not automatic archiving is enabled or disabled. If automatic archiving is disabled, the database administrator is responsible for manually archiving all filled groups.

For most systems, automatic archiving is chosen because the administrator does not have to watch for a group to become inactive and available for archiving. Furthermore, if automatic archiving is disabled and manual archiving is not performed fast enough, database operation can be suspended temporarily whenever LGWR is forced to wait for an inactive group to become available for reuse.

The manual archiving option is provided so that the database administrator can:

- Archive a group when automatic archiving has been stopped because of a problem (for example, the offline storage device specified as the archived redo log destination has experienced a failure or become full)
- Archive a group in a non-standard fashion (for example, archive one group to one offline storage device, the next group to a different offline storage device, and so on)
- Re-archive a group if the original archived version is lost or damaged

When a group is archived manually, the user process issuing the statement to archive a group actually performs the process of archiving the group. Even if the ARC*n* background process is present for the associated instance, it is the user process that archives the group of online redo log files.

Control Files

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, 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:

- *Oracle8i Recovery Manager User's Guide and Reference*
- *Oracle8i Backup and Recovery Guide*

for information about backing up a database's control file

Multiplexed Control Files

As with online redo log files, Oracle allows 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

Database Backups

You can use Oracle mirrored logs, Oracle mirrored control files, and archive logs to recover from media failure, but some or all of the data may not be available while recovery is proceeding. To achieve a higher level of recovery, Oracle recommends that you use operating system or hardware data redundancy for at least the datafiles and the control files. This will make sure that any one media failure will be recoverable while the system is fully available.

No matter what backup and recovery scheme you devise for an Oracle database, backups of the database's datafiles and control files are absolutely necessary as part of the strategy to safeguard against potential media failures that can damage these files. The following sections provide a conceptual overview of the different types of backups that can be made and their usefulness in different recovery schemes.

> **See Also:** *Oracle8i Backup and Recovery Guide* for guidelines for performing database backups

Whole Database Backups

A *whole database backup* is a backup of all datafiles and the control file that constitute an Oracle database. You can do this with Recovery Manager or by using operating system commands. You can take a whole database backup when the database is

shut down or while the database is open. You should not normally take a whole backup after an instance failure or other unusual circumstances.

Consistent Whole Backups vs. Inconsistent Whole Backups

A *clean shutdown* is a shutdown that does not follow a crash or a SHUTDOWN ABORT. Following a clean shutdown, all of the files that constitute a database are closed and consistent with respect to the current point in time. Thus, a whole backup taken after a shutdown can be used to recover to the point in time of that backup. A whole backup taken while the database is open is not consistent to a given point in time and must be recovered with the online and archived redo log files before the database can become available.

Backups and Archiving Mode

The datafiles obtained from a whole backup are useful in any type of media recovery scheme:

- If a database is operating in NOARCHIVELOG mode and a disk failure damages some or all of the files that constitute the database, the most recent consistent whole backup can be used to *restore* (not recover) the database.

  Because an archived redo log is not available to bring the database up to the current point in time, all database work performed since the backup must be repeated. Under special circumstances, a disk failure in NOARCHIVELOG mode can be fully recovered, but you should not rely on this.

- If a database is operating in ARCHIVELOG mode and a disk failure damages some or all of the files that constitute the database, the datafiles collected by the most recent whole backup can be used as part of database recovery.

  After restoring the necessary datafiles from the whole backup, database recovery can continue by applying archived and current online redo log files to bring the restored datafiles up to the current point in time.

In summary, if a database is operated in NOARCHIVELOG mode, a consistent whole database backup is the only method to partially protect the database against a disk failure. If a database is operating in ARCHIVELOG mode and the logs are available, either a consistent or an inconsistent whole database backup can be used to restore damaged files as part of database recovery from a disk failure.

Partial Database Backups

A *partial database backup* is any backup short of a whole backup, taken while the database is open or shut down. The following are all examples of partial database backups:

- A backup of all datafiles for an individual tablespace
- A backup of a single datafile
- A backup of a control file

Partial backups are only useful for a database operating in ARCHIVELOG mode. Because an archived redo log is present, the datafiles restored from a partial backup can be made consistent with the rest of the database during recovery procedures.

Datafile Backups

A partial backup includes only some of the datafiles of a database. Individual or collections of specific datafiles can be backed up independently of the other datafiles, online redo log files, and control files of a database. You can back up a datafile while it is offline or online.

Choosing whether to take online or offline datafile backups depends only on the availability requirements of the data—online datafile backups are the only choice if the data being backed up must always be available.

Control File Backups

Another form of a partial backup is a control file backup. Because a control file keeps track of the associated database's physical file structure, you should back up a database's control file every time a structural change is made to the database. Make a physical backup and a backup to a trace file.

> **Note:** Recovery Manager automatically backs up the control file in any backup that includes datafile 1, which contains the data dictionary.

Multiplexed control files safeguard against the loss of a single control file. However, if a disk failure damages the datafiles and incomplete recovery is desired, or a point-in-time recovery is desired, a backup of the control file that corresponds to the intended database structure should be used, not necessarily the current control file. Therefore, the use of multiplexed control files is not a substitute for control file backups taken every time the structure of a database is altered.

If you use Recovery Manager to restore the control file prior to incomplete or point-in-time recovery, Recovery Manager automatically restores the most suitable backup control file.

The Export and Import Utilities

Export and Import are utilities used to move Oracle data in and out of Oracle databases. Export is a utility that writes data from an Oracle database to operating system files in an Oracle database format. Export files store information about schema objects created for a database. Import is a utility that reads Export files and restores the corresponding information into an existing database. Although Export and Import are designed for moving Oracle data, they can be used also as a supplemental method of protecting data in an Oracle database.

See Also: *Oracle8i Utilities*

Read-Only Tablespaces and Backup

You can create backups of a read-only tablespace while the database is open. Immediately after making a tablespace read-only, you should back up the tablespace. As long as the tablespace remains read-only, there is no need to perform any further backups of it.

After you change a read-only tablespace to a read-write tablespace, you need to resume your normal backups of the tablespace, just as you do when you bring an offline read-write tablespace back online.

Bringing the datafiles of a read-only tablespace online does not make these files writable, nor does it cause the file header to be updated. Thus it is not necessary to perform a backup of these files, as is necessary when you bring a writable datafile back online.

Survivability

In the event of a power failure, hardware failure, or any other system-interrupting disaster, Oracle offers the *managed standby database* feature. The standby database is intended for sites where survivability and disaster recovery are of paramount importance. Another option is to use database replication.

See Also:

- Chapter 31, "Replication"
- *Oracle8i Standby Database Concepts and Administration*

Planning for Disaster Recovery

The only way to ensure rapid recovery from a system failure or other disaster is to plan carefully. You must have a set plan with detailed procedures. Whether you are implementing a standby database or you have a single database system, you must have a plan for what to do in the event of a catastrophic failure.

Managed Standby Database

Oracle provides the managed standby database feature to facilitate quick disaster recovery. Up to four standby systems can be maintained in a constant state of media recovery through the automatic shipping and application of log files archived at the primary site. In the event of a primary system failure, one of the standby systems can be activated, providing immediate system availability. Oracle provides commands and internal verifications for operations involved in the creation and maintenance of the standby systems, improving the reliability of the disaster recovery scheme.

A standby database uses the archived log information from the primary database, so it is ready to perform recovery and go online at any time. When the primary database archives its redo logs, the logs must be transferred to the remote site and applied to the standby database.

The managed standby database protects your data from extended outages such as power failures, or from physical disasters such as fire, floods, or earthquakes. Because the standby database is designed for disaster recovery, it ideally resides in a separate physical location from the primary database.

You can open the standby database read only. This allows you to use the database for reporting. When you open a standby database read only, redo logs are placed in a queue and are not applied. As soon as the database is returned to standby mode, the queued logs and newly arriving logs are applied.

See Also: *Oracle8i Standby Database Concepts and Administration* for information about creating and maintaining standby databases

Part IX

Distributed Databases and Replication

Part IX explains distributed database architecture and data replication across networks.

Part IX contains the following chapters:

30

Distributed Database Concepts

This chapter describes the basic concepts and terminology of Oracle's distributed database architecture. The chapter includes:

- Introduction to Distributed Database Architecture
- Database Links
- Distributed Database Administration
- Transaction Processing in a Distributed System
- Distributed Database Application Development
- National Language Support

> **See Also:** *Getting to Know Oracle8i* for information about features new to the current Oracle8*i* release

Introduction to Distributed Database Architecture

A *distributed database system* allows applications to access data from local and remote databases. In a *homogenous distributed system*, each database is an Oracle database. In a *heterogeneous distributed system*, at least one of the databases is a non-Oracle database. Distributed database uses a *client-server* architecture to process information requests.

This section contains the following topics:

Homogenous Distributed Database Systems

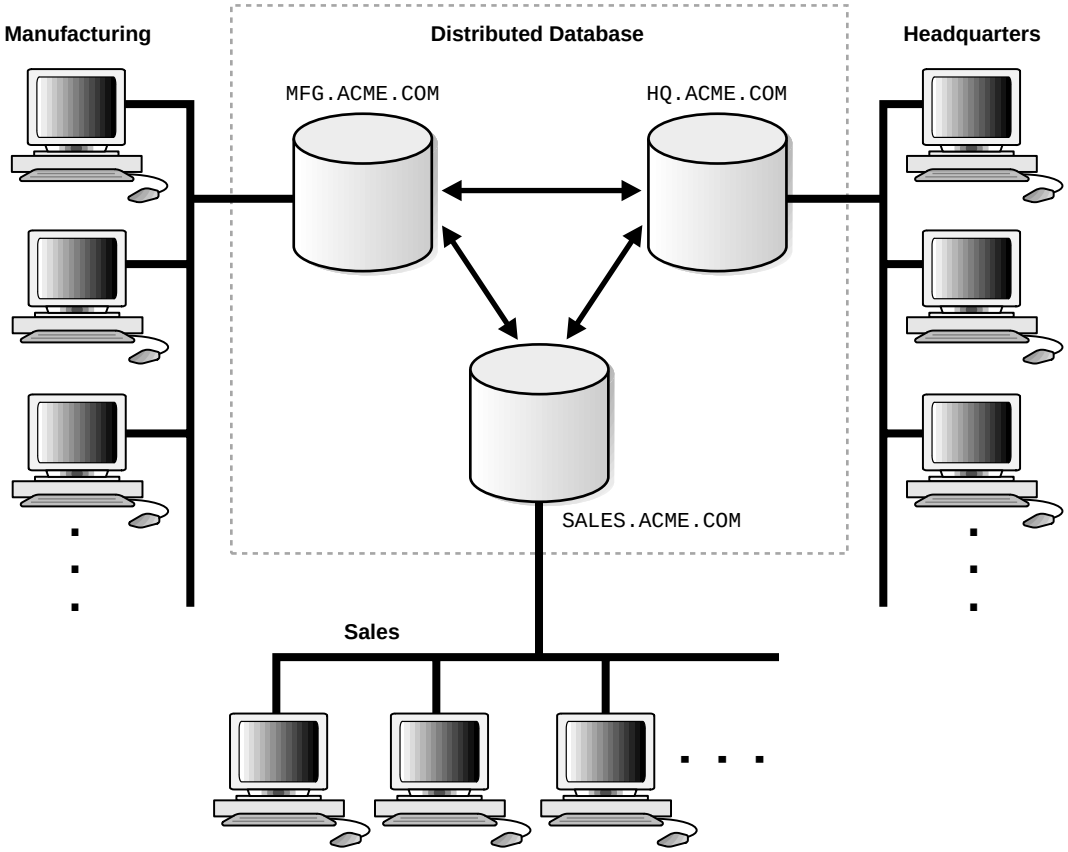
A *homogenous distributed database system* is a network of two or more Oracle databases that reside on one or more machines. Figure 30–1 illustrates a distributed system that connects three databases: HQ, MFG, and SALES. An application can simultaneously access or modify the data in several databases in a single distributed environment. For example, a single query on local database MFG can retrieve joined data from the PRODUCTS table on the local database and the DEPT table on the remote HQ database.

For a client application, the location and platform of the databases are transparent. You can also create *synonyms* for remote objects in the distributed system so that users can access them with the same syntax as local objects. For example, if you are connected to database MFG yet want to access data on database HQ, creating a synonym on MFG for the remote DEPT table allows you to issue this query:

```
SELECT * FROM dept;
```

In this way, a distributed system gives the appearance of native data access. Users on MFG do not have to know that the data they access resides on remote databases.

Figure 30–1 Homogeneous Distributed Database



An Oracle distributed database system can incorporate Oracle databases of different versions. All supported releases of Oracle can participate in a distributed database system. Nevertheless, the applications that work with the distributed database must understand the functionality that is available at each node in the system—for example, a distributed database application cannot expect an Oracle7 database to understand the object SQL extensions that are only available with Oracle8*i*.

Distributed Databases Vs. Distributed Processing

The terms *distributed database* and *distributed processing* are closely related, yet have distinct meanings:

Distributed Database	A set of databases in a distributed system that can appear to applications as a single data source.
Distributed Processing	The operations that occurs when an application distributes its tasks among different computers in a network. For example, a database application typically distributes front-end presentation tasks to client computers and allows a back-end database server to manage shared access to a database. Consequently, a distributed database application processing system is more commonly referred to as a *client-server* database application system.

Oracle distributed database systems employ a distributed processing architecture. For example, an Oracle server acts as a client when it requests data that another Oracle server manages.

Distributed Databases Vs. Replicated Databases

The terms *distributed database system* and *database replication* are related, yet distinct. In a *pure* (i.e., non-replicated) distributed database, the system manages a single copy of all data and supporting database objects. Typically, distributed database applications use distributed transactions to access both local and remote data and modify the global database in real-time.

Note: This book discusses only pure distributed databases.

The term *replication* refers to the operation of copying and maintaining database objects in multiple databases belonging to a distributed system. While replication relies on distributed database technology, database replication offers applications benefits that are not possible within a pure distributed database environment.

Most commonly, replication is used to improve local database performance and protect the availability of applications because alternate data access options exist. For example, an application may normally access a local database rather than a remote server to minimize network traffic and achieve maximum performance. Furthermore, the application can continue to function if the local server experiences a failure, but other servers with replicated data remain accessible.

> **See Also:** *Oracle8i Replication* for more information about Oracle's replication features

Heterogeneous Distributed Database Systems

In a *heterogeneous distributed database system*, at least one of the databases is a non-Oracle system. To the application, the heterogeneous distributed database system appears as a single, local, Oracle database; the local Oracle server hides the distribution and heterogeneity of the data.

The Oracle server accesses the non-Oracle system using Oracle8*i* Heterogeneous Services and a system-specific transparent gateway. For example, if you include a DB2 database in an Oracle distributed system, you need to obtain a DB2-specific transparent gateway so that the Oracle databases in the system can communicate with it.

> **See Also:** *Oracle8i Distributed Database Systems* for more information about Oracle's replication features

Heterogeneous Services

Heterogeneous Services are an integrated component within the Oracle8*i* server and the enabling technology for the current suite of Open Gateway products. Heterogeneous Services provide the common architecture and administration mechanisms for Oracle gateway products and other heterogeneous access facilities. Also, they provide upwardly compatible functionality for users of most of the earlier Oracle Open Gateway releases.

> **See Also:** *Oracle8i Distributed Database Systems* for an overview of heterogeneous services

Transparent Gateway Agents

For each non-Oracle system that you want to access, Heterogeneous Services require a *transparent gateway agent* to access the specified non-Oracle system. The transparent gateway agent facilitates communication between Oracle and non-Oracle databases and uses the Heterogeneous Services component in the Oracle server. The agent executes SQL, procedural, and transactional requests at the non-Oracle system on behalf of the Oracle server.

> **See Also:** *Oracle Open Gateway Installation and User's Guide* for detailed information on installation and configuration

Features

The features of the Heterogeneous Services include the following:

Feature	Purpose
Distributed Transactions	Allows a transaction to span both Oracle and non-Oracle systems, while still guaranteeing transaction consistency.
SQL Translations	Integrates data from non-Oracle systems into the Oracle environment as if the data is stored in one local database. SQL statements are transparently transformed into SQL statement understood by the non-Oracle system.
Procedural Access	Allows access to procedural systems, e.g., messaging from an Oracle8*i* server using PL/SQL remote procedure calls.
Data Dictionary Translations	Makes a non-Oracle system appear as an Oracle server. SQL statements containing references to Oracle's data dictionary tables are transformed into SQL statements containing references to a non-Oracle system's data dictionary tables.
Pass-Through SQL	Gives application programmers direct access to a non-Oracle system from an Oracle application using the non-Oracle system's SQL dialect.
Stored Procedure Access	Allows access to stored procedures in SQL-based non-Oracle systems as if they were PL/SQL remote procedures.
NLS Support	Supports multi-byte character sets, and translates character sets between a non-Oracle system and the Oracle8*i* server.
Multi-Threaded Agents	Takes advantage of your operating system's threading capabilities by reducing the number of required processes.
Agent Self-Registration	Automates the updating of Heterogeneous Services configuration data on remote hosts, ensuring correct operation over heterogeneous database links.

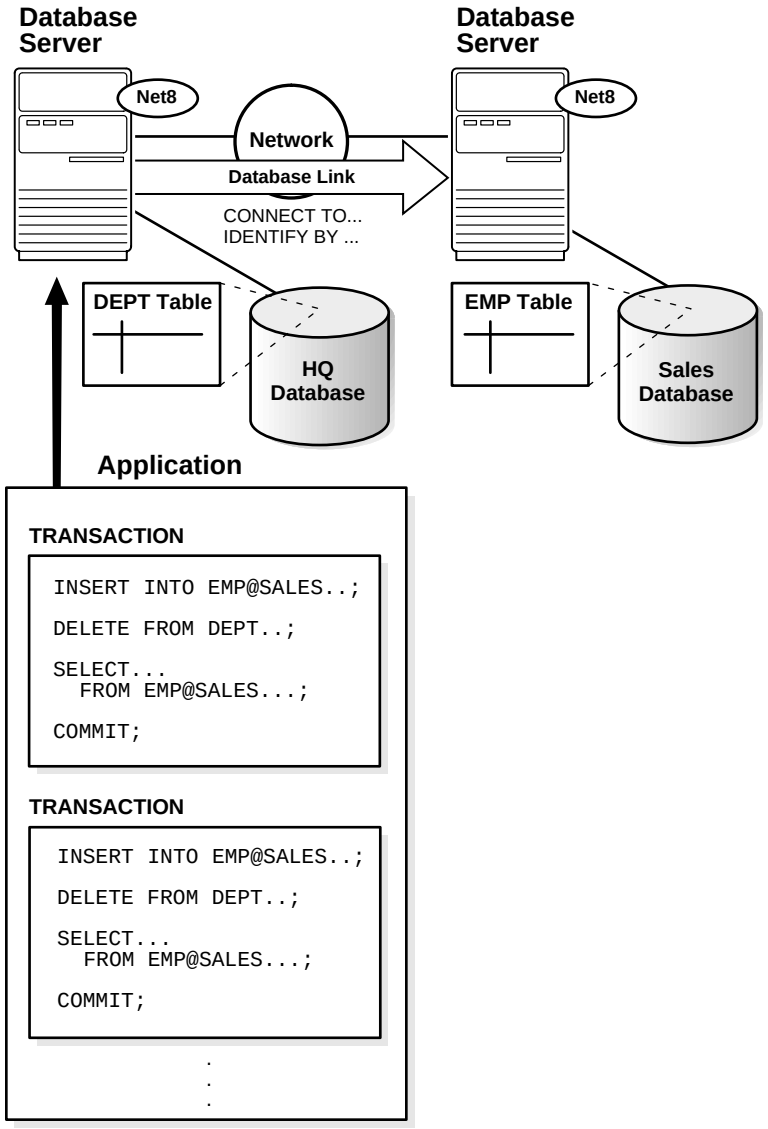
Note: Not all listed features are necessarily supported by your Heterogeneous Services agent or Oracle Gateway. See your system-specific documentation for supported features.

Client-Server Database Architecture

A database *server* is the Oracle software managing a database, and a *client* is an application that requests information from a server. Each computer in a network is a *node* that can host one or more databases. Each node in a distributed database system can act as a client, a server, or both, depending on the situation.

In Figure 30–2, the host for the HQ database is acting as a database server when a statement is issued against its local data (for example, the second statement in each transaction issues a statement against the local DEPT table), but is acting as a client when it issues a statement against remote data (for example, the first statement in each transaction is issued against the remote table EMP in the SALES database).

Figure 30–2 An Oracle Distributed Database System



Direct and Indirect Connections

A client can connect *directly* or *indirectly* to a database server. A direct connection occurs when a client connects to a server and accesses information from a database contained on that server. For example, if you connect to the HQ database and access the DEPT table on this database as in Figure 30–2, you can issue the following:

```
SELECT * FROM dept;
```

This query is direct because you are not accessing an object on a remote database.

In contrast, an indirect connection occurs when a client connects to a server and then accesses information contained in a database on a different server. For example, if you connect to the HQ database but access the EMP table on the remote SALES database as in Figure 30–2, you can issue the following:

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

This query is indirect because the object you are accessing is not on the database to which you are directly connected.

Database Links

The central concept in distributed database systems is a *database link*. A database links is a connection between two physical database servers that allows a client to access them as one logical database.

This section contains the following topics:

- What Are Database Links?
- Why Use Database Links?
- Global Database Names in Database Links
- Names for Database Links
- Types of Database Links
- Users of Database Links
- Creation of Database Links: Examples
- Schema Objects and Database Links
- Database Link Restrictions

What Are Database Links?

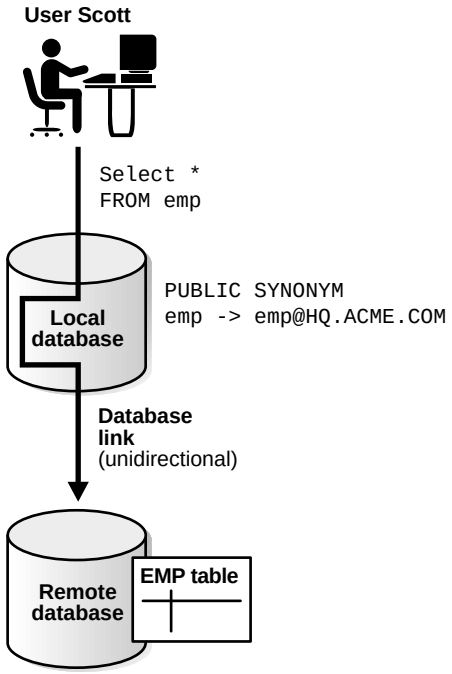
A database link is a pointer that defines a one-way communication path from an Oracle database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A database link connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, they must define a link that is stored in the data dictionary of database B.

A database link connection allows local users to access data on a remote database. For this connection to occur, each database in the distributed system must have a unique *global database name* in the network domain. The global database name uniquely identifies a database server in a distributed system.

Figure 30–3 shows an example of user SCOTT accessing the EMP table on the remote database with the global name HQ.ACME.COM:

Figure 30–3 Database Link



Database links are either *private* or *public*. If they are private, only the user who created the link has access; if they are public, all database users have access.

One principal difference among database links is the way that connections to a remote database are made. Users accessing a remote database using a:

- *Connected user link* connect as themselves, which means that they must have an account on the remote database with the same username as their account on the local database.
- *Fixed user link* connect using the username and password referenced in the link. For example, if JANE uses a fixed user link that connects to the HR database with the username and password SCOTT/TIGER, she connects as SCOTT. JANE has all the privileges in HR granted to SCOTT directly, and all the default roles that Scott has been granted in the HR database.
- *Current user link* connect as a global user. A local user can connect as a global user in the context of a stored procedure—without storing the global user's password in a link definition. For example, JANE can access a procedure that

SCOTT wrote, accessing SCOTT's account and SCOTT's schema on the HR database. Current user links are an aspect of the Oracle Advanced Security option.

Create database links using the CREATE DATABASE LINK statement. After a link is created, you can use it to specify schema objects in SQL statements.

What Are Shared Database Links?

A shared database link is a link between a local server process and the remote database. The link is shared because multiple client processes can use the same link simultaneously.

When a local database is connected to a remote database through a database link, either database can run in dedicated or multi-threaded server (MTS) mode. The following table illustrates the possibilities:

Local Database Mode	Remote Database Mode
Dedicated	Dedicated
Dedicated	Multi-threaded
Multi-threaded	Dedicated
Multi-threaded	Multi-threaded

A shared database link can exist in any of these four configurations. Shared links differ from standard database links in the following ways:

- Different users accessing the same schema object via a database link can share a network connection.
- When a user needs to establish a connection to a remote server from a particular server process, the process can reuse connections already established to the remote server. The reuse of the connection can occur if the connection was established on the same server process with the same database link—possibly in a different session. In a non-shared database link, a connection is not shared across multiple sessions.
- When you use a shared database link in an MTS configuration, a network connection is established directly out of the shared server process in the local server. For a non-shared database link on a local multi-threaded server, this connection would have been established through the local dispatcher, requiring context switches for the local dispatcher, and requiring data to go through the dispatcher.

See Also: *Net8 Administrator's Guide* for information about the multi-threaded server option

Why Use Database Links?

The great advantage of database links is that they allow users to access another user's objects in a remote database so that they are bounded by the privilege set of the object's owner. In other words, a local user can access a link to a remote database without having to be a user on the remote database.

For example, assume that employees submit expense reports to Accounts Payable (A/P), and further suppose that a user using an A/P application needs to retrieve information about employees from the HR database. The A/P users should be able to connect to the HR database and execute a stored procedure in the remote HR database that retrieves the desired information. The A/P users should not need to be HR database users to do their jobs; they should only be able to access HR information in a controlled way as limited by the procedure.

Database links allow you to grant limited access on remote databases to local users. By using current user links, you can create centrally managed global users whose password information is hidden from both administrators *and* non-administrators. For example, A/P users can access the HR database as SCOTT, but unlike fixed user links, SCOTT's credentials are not stored where database users can see them.

By using fixed user links, you can create non-global users whose password information is stored in unencrypted form in the LINK$ data dictionary table. Fixed user links are easy to create and require low overhead because there are no SSL or directory requirements, but a security risk results from the storage of password information in the data dictionary.

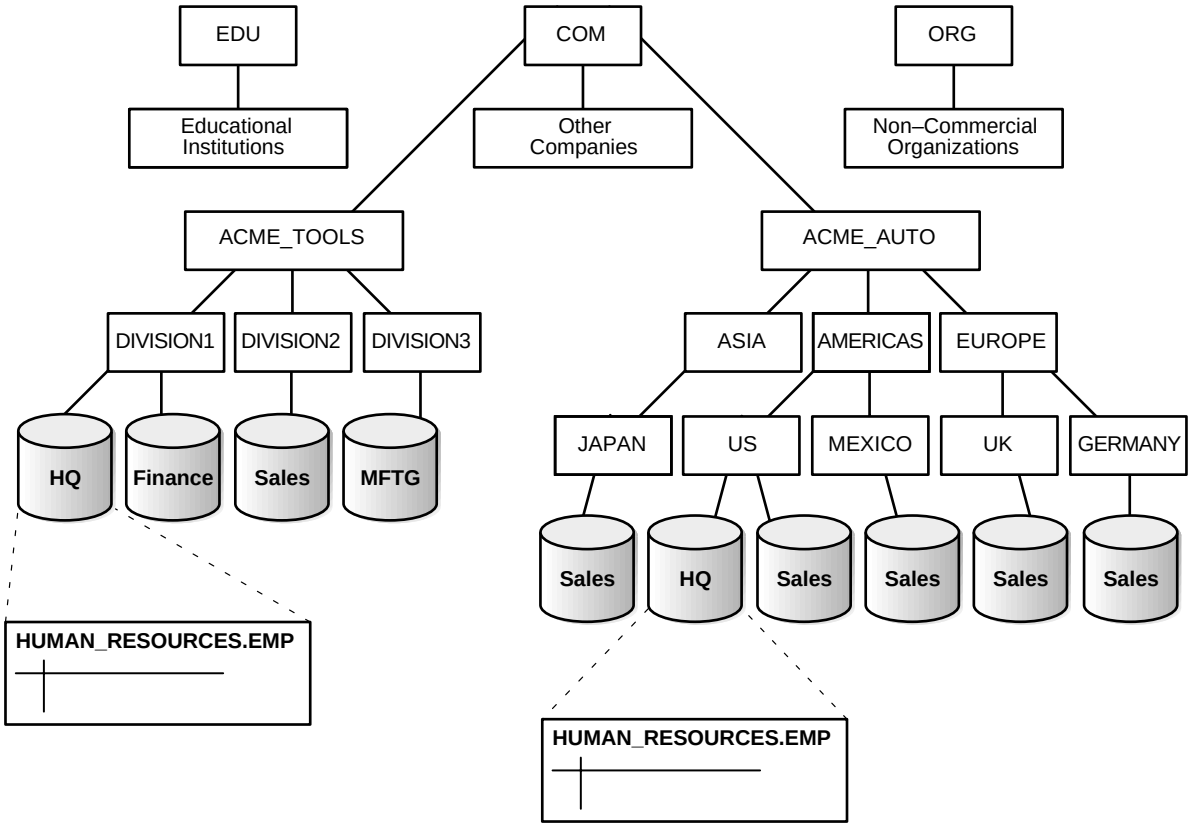
See Also: *Oracle8i Distributed Database Systems* for an explanation of database link users and how to hide passwords from non-administrators

Global Database Names in Database Links

To understand how a database link works, you must first understand what a global database name is. Each database in a distributed database is uniquely identified by its global database name. Oracle forms a database's global database name by prefixing the database's network domain, specified by the DB_DOMAIN initialization parameter at database creation, with the individual database name, specified by the DB_NAME initialization parameter.

For example, Figure 30–4 illustrates a representative hierarchical arrangement of databases throughout a network.

Figure 30–4 Network Directories and Global Database Names



The name of a database is formed by starting at the leaf of the tree and following a path to the root. For example, the MFTG database is in DIVISION3 of the ACME_TOOLS branch of the COM domain. The global database name for MFTG is created by concatenating the nodes in the tree as follows:

```
MFTG.DIVISION3.ACME_TOOLS.COM
```

While several databases can share an individual name, each database must have a unique global database name. For example, the network domains US.AMERICAS.ACME_AUTO.COM and UK.EUROPE.ACME_AUTO.COM each

contain a SALES database. The global database naming system distinguishes the SALES database in the AMERICAS division from the SALES database in the EUROPE division as follows:

```
SALES.US.AMERICAS.ACME_AUTO.COM
SALES.UK.EUROPE.ACME_AUTO.COM
```

> **See Also:** *Oracle8i Distributed Database Systems* to learn how to specify and change global database names

Names for Database Links

Typically, a database link has the same name as the global database name of the remote database it references. For example, if the global database name of a database is SALES.US.ORACLE.COM, then the database link is also called SALES.US.ORACLE.COM.

When you set the initialization parameter GLOBAL_NAMES to TRUE, Oracle ensures that the name of the database link is the same as the global database name of the remote database. For example, if the global database name for HQ is HQ.ACME.COM, and GLOBAL_NAMES is TRUE, then the link name must be called HQ.ACME.COM. Note that Oracle checks the domain part of the global database name as stored in the data dictionary, *not* the DB_DOMAIN setting in the `init.ora` file (see *Oracle8i Distributed Database Systems*).

If you set the initialization parameter GLOBAL_NAMES to FALSE, you are not required to use global naming. You can then name the database link whatever you want. For example, you can name a database link to HQ.ACME.COM as FOO.

> **Note:** Oracle Corporation recommends that you use global naming because many useful features, including Oracle Advanced Replication, require global naming be enforced.

After you have enabled global naming, database links are essentially transparent to users of a distributed database because the name of a database link is the same as the global name of the database to which the link points. For example, the following statement creates a database link in the local database to remote database SALES:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com USING 'sales1';
```

> **See Also:** *Oracle8i Reference* for more information about specifying the initialization parameter GLOBAL_NAMES

Types of Database Links

Oracle allows you to create *private*, *public*, and *global* database links. These basic link types differ according to which users are allowed access to the remote database:

Type	Owner	Description
Private	User who created the link. Access ownership data via DBA_DB_LINKS or ALL_DB_LINKS.	Creates link in a specific schema of the local database. Only the owner of a private database link or PL/SQL subprograms in the schema can use a private database link to access database objects in the corresponding remote database.
Public	User called PUBLIC. Access ownership data via DBA_DB_LINKS or ALL_DB_LINKS.	Creates a database-wide link. All users and PL/SQL subprograms in the database can use the link to access database objects in the corresponding remote database.
Global	User called PUBLIC. Access ownership data via DBA_DB_LINKS or ALL_DB_LINKS.	Creates a network-wide link. When an Oracle network uses Oracle Names, the names servers in the system automatically create and manage global database links for every Oracle database in the network. All users and PL/SQL subprograms in any database can use a global link to access database objects in the corresponding remote database.

Determining the type of database links to employ in a distributed database depends on the specific requirements of the applications using the system. Consider these advantages and disadvantages:

Private Database Link	This link is more secure than a public or global link, because only the owner of the private link, or subprograms within the same schema, can use the link to access the remote database.
Public Database Link	When many users require an access path to a remote Oracle database, you can create a single public database link for all users in a database.
Global Database Link	When an Oracle network uses Oracle Names, an administrator can conveniently manage global database links for all databases in the system. Database link management is centralized and simple.

See Also: *Oracle8i Distributed Database Systems* to learn how to create different types of database links sand to learn how to access information about links

Users of Database Links

When creating the link, you determine which user should connect to the remote database to access the data. The following table explains the differences between the categories of users involved in database links:

User Type	Meaning	Sample Link Creation Syntax
Connected user	A local user accessing a database link in which no fixed username and password have been specified. If SYSTEM accesses a public link in a query, then the connected user is SYSTEM, and Oracle connects to the SYSTEM schema in the remote database. **Note:** A connected user does not have to be the user who created the link, but is any user who is accessing the link.	CREATE PUBLIC DATABASE LINK hq USING 'hq';
Current user	A global user in a CURRENT USER database link. The global user must be authenticated by an X.509 certificate and a private key and be a user on both databases involved in the link. Current user links are an aspect of the Oracle Advanced Security (OAS) option. **See Also:** *Oracle Advanced Security Administrator's Guide* for more information about global security.	CREATE PUBLIC DATABASE LINK hq CONNECT TO CURRENT_USER using 'hq';
Fixed user	A user whose username/password is part of the link definition. If a link includes a fixed user, then the fixed user's username and password are used to connect to the remote database.	CREATE PUBLIC DATABASE LINK hq CONNECT TO jane IDENTIFIED BY doe USING 'hq';

See Also: *Oracle8i Distributed Database Systems* to learn how to specify users where creating links

Connected User Database Links

Connected user links have no connect string associated with them. The advantage of a connected user link is that a user referencing the link connects to the remote database as the same user. Furthermore, because no connect string is associated with the link, no password is stored in clear text in the data dictionary.

Connected user links have some disadvantages. Because these links require users to have accounts and privileges on the remote databases to which they are attempting

to connect, they require more privilege administration for administrators. Also, giving users more privileges than they need violates the fundamental security concept of least privilege: users should only be given the least privilege they need to perform their jobs.

The ability to use connected user database link depends on several factors, chief among them whether the user is authenticated by Oracle using a password, or externally authenticated by the operating system or a network authentication service. If the user is externally authenticated, then the ability to use a connected user link also depends on whether the remote database accepts remote authentication of users, which is set by the REMOTE_OS_AUTHENT initialization parameter.

The REMOTE_OS_AUTHENT parameter operates as follows:

If REMOTE_OS_AUTHENT is	Then
TRUE for the remote database	An externally-authenticated user can connect to the remote database using a connected user database link.
FALSE for the remote database	An externally-authenticated user cannot connect to the remote database using a connected user database link unless a secure protocol is used or a network authentication service supported by the Oracle Advanced Security option is used.

Fixed User Database Links

A benefit of a named link is that it connects a user in a primary database to a remote database with the security context of the user in the connect string. For example, local user JOE can create a public database link in JOE's schema that specifies the fixed user SCOTT with password TIGER. If JANE uses the fixed user link in a query, then JANE is the user on the local database, but she connects to the remote database as SCOTT/TIGER.

Fixed user links have a username and password associated with the connect string. The username and password are stored in unencrypted form in the data dictionary in the LINK$ table. This fact creates a possible security weakness of fixed user database links: a user with the SELECT ANY TABLE privilege has access to the data dictionary so long as the O7_DICTIONARY_ACCESSIBILITY initialization parameter is set to TRUE, and thus the authentication associated with a fixed user is compromised.

Note: The default value for O7_DICTIONARY_ACCESSIBILITY is FALSE.

For an example of this security problem, assume that JANE does not have privileges to use a private link that connects to the HR database as SCOTT/TIGER, but has SELECT ANY TABLE privilege on a database in which the O7_DICTIONARY_ACCESSIBILITY initialization parameter is set to TRUE. She can select from LINK$ and read that the connect string to HR is SCOTT/TIGER. If JANE has an account on the host on which HR resides, then she can connect to the host and then connect to HR as SCOTT using the password TIGER. She will have all SCOTT's privileges if she connects locally and any audit records will be recorded as if she were SCOTT.

Current User Database Links

Current user database links make use of a global user. A global user must be authenticated by an X.509 certificate and a private key and be a user on both databases involved in the link.

The user invoking the CURRENT_USER link does not have to be a global user. For example, if JANE is authenticated by password to the Accounts Payable database, she can access a stored procedure to retrieve data from the HR database. The procedure uses a current user database link, which connects her to HR as global user SCOTT. SCOTT is a global user and thereby authenticated through a certificate and private key over SSL, but JANE is not.

Note that current user database links have these consequences:

- If the current user database link is *not* accessed from within a stored object, then the current user is the same as the connected user accessing the link. For example, if SCOTT issues a SELECT statement through a current user link, then the current user is SCOTT.
- When executing a stored object such as a procedure, view, or trigger that accesses a database link, the current user is the user that *owns* the stored object, and not the user that *calls* the object. For example, if JANE calls procedure SCOTT.P (created by SCOTT), and a current user link appears *within* the called procedure, then SCOTT is the current user of the link.
- If the stored object is an invoker-rights function, procedure, or package, the invoker's authorization ID is used to connect as a remote user. For example, if user JANE calls procedure SCOTT.P (an invoker-rights procedure created by

SCOTT), and the link appears inside procedure SCOTT.P, then JANE is the current user.

> **See Also:** "Distributed Database Security" on page 30-24 for more information about security issues relating to database links

Creation of Database Links: Examples

Create database links using the CREATE DATABASE LINK statement. The table gives examples of SQL statements that create database links in a *local* database to the *remote* SALES.US.AMERICAS.ACME_AUTO.COM database:

SQL Statement	Connects To Database	Connects As	Link Type
`CREATE DATABASE LINK sales.us.americas.acme_auto.com USING 'sales_us';`	SALES using net service name SALES_US	Connected user	Private connected user
`CREATE DATABASE LINK foo CONNECT TO CURRENT_USER USING 'am_sls';`	SALES using service name AM_SLS	Current global user	Private current user
`CREATE DATABASE LINK sales.us.americas.acme_auto.com CONNECT TO scott IDENTIFIED BY tiger USING 'sales_us';`	SALES using net service name SALES_US	SCOTT using password TIGER	Private fixed user
`CREATE PUBLIC DATABASE LINK sales CONNECT TO scott IDENTIFIED BY tiger USING 'rev';`	SALES using net service name REV	SCOTT using password TIGER	Public fixed user
`CREATE SHARED PUBLIC DATABASE LINK sales.us.americas.acme_auto.com CONNECT TO scott IDENTIFIED BY tiger AUTHENTICATED BY anupam IDENTIFIED BY bhide USING 'sales';`	SALES using net service name SALES	SCOTT using password TIGER, authenticated as ANUPAM using password BHIDE	Shared public fixed user

> **See Also:**
>
> - *Oracle8i Distributed Database Systems* to learn how to create links
> - *Oracle8i SQL Reference* for CREATE DATABASE LINK syntax

Schema Objects and Database Links

After you have created a database link, you can execute SQL statements that access objects on the remote database. For example, to access remote object EMP using database link FOO, you can issue:

```
SELECT * FROM emp@foo;
```

Constructing properly formed object names using database links is an essential aspect of data manipulation in distributed systems.

Naming of Schema Objects Using Database Links

Oracle uses the global database name to name the schema objects globally using the following scheme:

schema.schema_object@global_database_name

where:

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_object	is a logical data structure like a table, index, view, synonym, procedure, package, or a database link.
global_database_name	is the name that uniquely identifies a remote database. This name must be the same as the concatenation of the remote database's initialization parameters DB_NAME and DB_DOMAIN, unless the parameter GLOBAL_NAMES is set to FALSE, in which case any name is acceptable.

For example, using a database link to database SALES.DIVISION3.ACME.COM, a user or application can reference remote data as follows:

```
SELECT * FROM scott.emp@sales.division3.acme.com;  # emp table in scott's schema
SELECT loc FROM scott.dept@sales.division3.acme.com;
```

If GLOBAL_NAMES is set to FALSE, then you can use any name for the link to SALES.DIVISION3.ACME.COM. For example, you can call the link FOO. Then, you can access database as follows:

```
SELECT name FROM scott.emp@foo;  # link name different from global name
```

Synonyms for Schema Objects

Oracle allows you to create *synonyms* so that you can hide the database link name from the user. A synonym allows access to a table on a remote database using the same syntax that you would use to access a table on a local database. For example, assume you issue the following query against a table in a remote database:

```
SELECT * FROM emp@hq.acme.com;
```

You can create the synonym EMP for EMP@HQ.ACME.COM so that you can issue the following query instead to access the same data:

```
SELECT * FROM emp;
```

> **See Also:** *Oracle8i Distributed Database Systems* to learn how to create synonyms for objects specified using database links

Schema Object Name Resolution

To resolve application references to schema objects (a process called *name resolution*), Oracle forms object names hierarchically. For example, Oracle guarantees that each schema within a database has a unique name, and that within a schema each object has a unique name. As a result, a schema object's name is always unique within the database. Furthermore, Oracle resolves application references to an object's local name.

In a distributed database, a schema object such as a table is accessible to all applications in the system. Oracle extends the hierarchical naming model with global database names to effectively create *global object names* and resolve references to the schema objects in a distributed database system. For example, a query can reference a remote table by specifying its fully qualified name, including the database in which it resides.

For example, assume that you connect to the local database as user SYSTEM:

```
CONNECT system/manager@sales1
```

You then issue the following statements using database link HQ.ACME.COM to access objects in the SCOTT and JANE schemas on remote database HQ:

```
SELECT * FROM scott.emp@hq.acme.com;
INSERT INTO jane.accounts@hq.acme.com (acc_no, acc_name, balance)
  VALUES (5001, 'BOWER', 2000);
UPDATE jane.accounts@hq.acme.com
  SET balance = balance + 500;
DELETE FROM jane.accounts@hq.acme.com
  WHERE acc_name = 'BOWER';
```

Database Link Restrictions

You *cannot* perform the following operations via database links:

- Grant privileges on remote objects.
- Execute DESCRIBE operations on some remote objects. The following remote objects, however, support DESCRIBE operations:
  - Tables
  - Views
  - Procedures
  - Functions
- ANALYZE remote objects.
- Define or enforce referential integrity.
- Grant roles to users in a remote database.
- Obtain non-default roles on a remote database. For example, if JANE connects to the local database and executes a stored procedure that uses a fixed user link connecting as SCOTT, JANE receives SCOTT's default roles on the remote database. Jane cannot issue SET ROLE to obtain a non-default role.
- Execute hash query joints that use MTS connections.
- Use a current user link without authentication through SSL or NT native authentication.

Distributed Database Administration

The following sections explain some of the topics relating to database management in an Oracle distributed database system:

- Site Autonomy
- Distributed Database Security
- Auditing Database Links
- Administration Tools

> **See Also:** *Oracle8i Distributed Database Systems* to learn how to administer homogenous systems and to learn how to administer heterogeneous systems

Site Autonomy

Site autonomy means that each server participating in a distributed database is administered independently from all other databases. Although several databases can work together, each database is a separate repository of data that is managed individually. Some of the benefits of site autonomy in an Oracle distributed database include:

- Nodes of the system can mirror the logical organization of companies or groups that need to maintain independence.
- Local administrators control corresponding local data. Therefore, each database administrator's domain of responsibility is smaller and more manageable.
- Independent failures are less likely to disrupt other nodes of the distributed database. No single database failure need halt all distributed operations or be a performance bottleneck.
- Administrators can recover from isolated system failures independently from other nodes in the system.
- A data dictionary exists for each local database—a global catalog is not necessary to access local data.
- Nodes can upgrade software independently.

Although Oracle allows you to manage each database in a distributed database system independently, you should not ignore the global requirements of the system. For example, you may need to:

- Create additional user accounts in each database to support the links that you create to facilitate server-to-server connections.
- Set additional initialization parameters such as DISTRIBUTED_LOCK_TIMEOUT, DISTRIBUTED_TRANSACTIONS, COMMIT_POINT_STRENGTH, and so forth.

Distributed Database Security

Oracle supports all of the security features that are available with a non-distributed database environment for distributed database systems, including:

- Password authentication for users and roles
- Some types of external authentication for users and roles including:
  - Kerberos version 5 for connected user links

  - DCE for connected user links
- Login packet encryption for client-to-server and server-to-server connections

The following sections explain some additional topics to consider when configuring an Oracle distributed database system:

- Authentication Through Database Links
- Supporting User Accounts and Roles
- Centralized User and Privilege Management
- Data Encryption

> **See Also:** *Oracle Advanced Security Administrator's Guide* for more information about external authentication

Authentication Through Database Links

Database links are either *private* or *public*, *authenticated* or *non-authenticated*. You create public links by specifying the PUBLIC keyword in the link creation statement. For example, you can issue:

```
CREATE PUBLIC DATABASE LINK foo USING 'sales';
```

You create authenticated links by specifying the CONNECT TO clause, AUTHENTICATED BY clause, or both clauses together in the database link creation statement. For example, you can issue:

```
CREATE DATABASE LINK sales CONNECT TO scott IDENTIFIED BY tiger USING 'sales';
CREATE SHARED PUBLIC DATABASE LINK sales CONNECT TO mick IDENTIFIED BY jagger
     AUTHENTICATED BY david IDENTIFIED BY bowie USING 'sales';
```

This table describes how users access the remote database through the link:

Link Type	Authenticated?	Security Access
Private	No	When connecting to the remote database, Oracle uses security information (userid / password) taken from the local session. Hence, the link is a connected user database link. Passwords must be synchronized between the two databases.

Link Type	Authenticated?	Security Access
Private	Yes	The userid / password is taken from the link definition rather than from the local session context. Hence, the link is a fixed user database link. This configuration allows passwords to be different on the two databases, but the local database link password must match the remote database password. The password is stored in clear text on the local system catalog, adding a security risk.
Public	No	Works the same as a private non-authenticated link, except that all users can reference this pointer to the remote database.
Public	Yes	All users on the local database can access the remote database and all use the same userid / password to make the connection. Also, the password is stored in clear text in the local catalog, so you can see the password if you have sufficient privileges in the local database.

Authentication Without Passwords

When using a connected user or current user database link, you can use an external authentication source such as Kerberos to obtain *end-to-end security*. In end-to-end authentication, credentials are passed from server to server and can be authenticated by an database server belonging to the same domain. For example, if JANE is authenticated externally on a local database, and wants to use a connected user link to connect as herself to a remote database, the local server passes the security ticket to the remote database.

Supporting User Accounts and Roles

In a distributed database system, you must carefully plan the user accounts and roles that are necessary to support applications using the system. Note that:

- The user accounts necessary to establish server-to-server connections must be available in all databases of the distributed database system.
- The roles necessary to make available application privileges to distributed database application users must be present in all databases of the distributed database system.

As you create the database links for the nodes in a distributed database system, determine which user accounts and roles each site needs to support server-to-server connections that use the links.

In a distributed environment, users typically require access to many network services. When you must configure separate authentications for each user to access each network service, security administration can become unwieldy, especially for large systems.

> **See Also:** *Oracle8i Distributed Database Systems* for more information about the user accounts that must be available to support different types of database links in the system

Centralized User and Privilege Management

Oracle provides different ways for you to manage the users and privileges involved in a distributed system. For example, you have these options:

- Enterprise user management. You can use create global users that are authenticated through SSL, then manage these users and their privileges in a directory through an independent enterprise directory service.
- Network authentication service. This common technique simplifies security management for distributed environments. You can use the Net8 Oracle Advanced Security option to enhance Net8 and the security of an Oracle distributed database system. Windows NT native authentication is an example of a non-Oracle authentication solution.

  See Also:

  - *Net8 Administrator's Guide*
  - *Oracle Advanced Security Administrator's Guide*

  for more information about global user security

Schema-Dependent Global Users One option for centralizing user and privilege management is to create the following:

- A global user in a centralized directory.
- A user in every database that the global user must connect to.

For example, you can create a global user called FRED with the following SQL statement:

```
CREATE USER fred IDENTIFIED GLOBALLY AS 'CN=fred adams,O=Oracle,C=England';
```

This solution allows a single global user to be authenticated by a centralized directory.

The schema-dependent global user solution has the consequence that you must create a user called FRED on every database that this user must access. Because most users need permission to access an application schema but do not need their own schemas, the creation of a separate account in each database for every global user creates significant overhead. Because of this problem, Oracle also supports schema-independent users, which are global users that an access a single, generic schema in every database.

Separation of Global Users from Database Schema Oracle8*i* supports functionality that allows a global user to be centrally managed by an enterprise directory service. Users who are managed in the directory are called *enterprise users*. This directory contains information about:

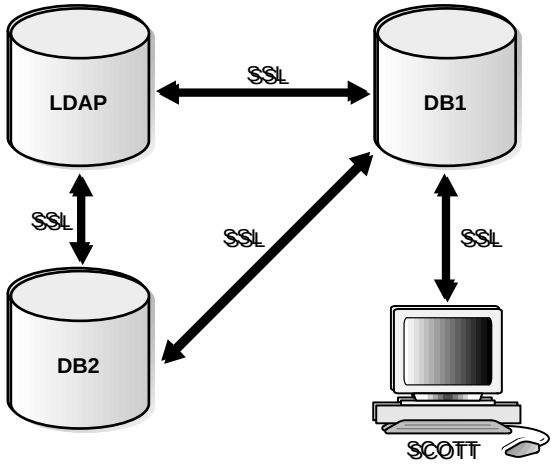
- Which databases in a distributed system an enterprise user can access
- Which role on each database an enterprise user can use
- Which schema on each database an enterprise user can connect to

The administrator of each database is not required to create a global user account for each enterprise user on each database to which the enterprise user needs to connect. Instead, multiple enterprise users can connect to the same database schema, called a *shared schema*.

For example, suppose JANE, BILL, and SCOTT all use a human resources application. The HR application objects are all contained in the GUEST schema on the HR database. In this case, you can create a local global user account to be used as a shared schema. This global username, that is, shared schema name, is GUEST. JANE, BILL, and SCOTT are all created as enterprise users in the directory service. They are also mapped to the HRAPP schema in the directory, and can be assigned different authorizations in the HR application.

Figure 30–5 illustrates an example of global user security using the enterprise directory service:

Figure 30–5 Global User Security



Assume that the enterprise directory service contains the following information on enterprise users for DB1 and DB2:

Database	Role	Schema	Enterprise Users
DB1	clerk1	guest	bill
			scott
DB2	clerk2	guest	jane
			scott

Also, assume that the local administrators for DB1 and DB2 have issued statements as follows:

Database	CREATE Statements

DB1	`CREATE USER guest IDENTIFIED GLOBALLY AS '';`
	`CREATE ROLE clerk1 GRANT select ON emp;`
	`CREATE PUBLIC DATABASE LINK db2_link CONNECT AS CURRENT_USER USING 'db2';`

DB2	`CREATE USER guest IDENTIFIED GLOBALLY;` `CREATE ROLE clerk2 GRANT select ON dept;`

Assume that enterprise user SCOTT requests a connection to local database DB1 in order to execute a distributed transaction involving DB2. The following steps occur (not necessarily in this exact order):

1. Enterprise user SCOTT and database DB1 mutually authenticate one another using SSL.
2. SCOTT issues the following statement:

```
SELECT e.ename, d.loc
FROM emp e, dept@db2_link d
WHERE e.deptno=d.deptno
```

3. Databases DB1 and DB2 mutually authenticate one another using SSL.
4. DB1 queries the enterprise directory service to determine whether enterprise user SCOTT has access to DB1, and discovers SCOTT can access local schema GUEST using role CLERK1.
5. Database DB2 queries the enterprise directory service to determine whether enterprise user SCOTT has access to DB2, and discovers SCOTT can access local schema GUEST using role CLERK2.
6. Enterprise user SCOTT logs into DB2 to schema GUEST with role CLERK2 and issues a SELECT to obtain the required information and transfer it to DB1.
7. DB1 receives the requested data from DB2 and returns it to the client SCOTT.

See Also:

- *Net8 Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*

for more information about enterprise user security

Data Encryption

The Oracle Advanced Security option also enables Net8 and related products to use network data encryption and checksumming so that data cannot be read or altered. It protects data from unauthorized viewing by using the RSA Data Security RC4 or the Data Encryption Standard (DES) encryption algorithm.

To ensure that data has not been modified, deleted, or replayed during transmission, the security services of the Oracle Advanced Security option can generate a cryptographically secure message digest and include it with each packet sent across the network.

See Also:

- *Net8 Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*

for more information about these and other features of the Oracle Advanced Security option

Auditing Database Links

You must always perform auditing operations locally. That is, if a user acts in a local database and accesses a remote database through a database link, the local actions are audited in the local database, and the remote actions are audited in the remote database—provided that appropriate audit options are set in the respective databases.

The remote database cannot determine whether a successful connect request and subsequent SQL statements come from another server or from a locally connected client. For example, assume the following:

- Fixed user link HR.ACME.COM connects local user JANE to the remote HR database as remote user SCOTT
- SCOTT is audited on the remote database

Actions performed during the remote database session are audited as if SCOTT were connected locally and performing the same actions there. You must set audit options in the remote database to capture the actions of the username—in this case, SCOTT—embedded in the link if the desired effect is to audit what JANE is doing in the remote database.

Note: You can audit the global username for global users.

You cannot set local auditing options on remote objects. Therefore, you cannot audit use of a database link, although access to remote objects can be audited on the remote database.

Administration Tools

The database administrator has several choices for tools to use when managing an Oracle distributed database system:

- Enterprise Manager

- Third-Party Administration Tools
- SNMP Support

Enterprise Manager

Enterprise Manager is Oracle's database administration tool. The graphical component of Enterprise Manager (Enterprise Manager/GUI) allows you to perform database administration tasks with the convenience of a graphical user interface (GUI). The non-graphical component of Enterprise Manager provides a command line interface.

Enterprise Manager provides administrative functionality via an easy-to-use interface. You can use Enterprise Manager to:

- Perform traditional administrative tasks, such as database startup, shutdown, backup, and recovery. Rather than manually entering the SQL statements to perform these tasks, you can use Enterprise Manager's graphical interface to execute the commands quickly and conveniently by pointing and clicking with the mouse.
- Concurrently perform multiple tasks. Because you can open multiple windows simultaneously in Enterprise Manager, you can perform multiple administrative and non-administrative tasks concurrently.
- Administer multiple databases. You can use Enterprise Manager to administer a single database or to simultaneously administer multiple databases.
- Centralize database administration tasks. You can administer both local and remote databases running on any Oracle platform in any location worldwide. In addition, these Oracle platforms can be connected by any network protocol(s) supported by Net8.
- Dynamically execute SQL, PL/SQL, and Enterprise Manager statements. You can use Enterprise Manager to enter, edit, and execute statements. Enterprise Manager also maintains a history of statements executed.

  Thus, you can re-execute statements without retyping them, a particularly useful feature if you need to execute lengthy statements repeatedly in a distributed database system.
- Perform administrative tasks using Enterprise Manager's line-mode interface when a graphical user interface is unavailable or undesirable.
- Manage security features such as global users, global roles, and the enterprise directory service.

Third-Party Administration Tools

Currently more than 60 companies produce more than 150 products that help manage Oracle databases and networks, providing a truly open environment.

SNMP Support

Besides its network administration capabilities, Oracle *Simple Network Management Protocol* (*SNMP*) support allows an Oracle server to be located and queried by any SNMP-based network management system. SNMP is the accepted standard underlying many popular network management systems such as:

- HP's OpenView
- Digital's POLYCENTER Manager on NetView
- IBM's NetView / 6000
- Novell's NetWare Management System
- SunSoft's SunNet Manager

> **See Also:** *Oracle SNMP Support Reference Guide* for more information about SNMP

Transaction Processing in a Distributed System

A transaction is a logical unit of work constituted by one or more SQL statements executed by a single user. A transaction begins with the user's first executable SQL statement and ends when it is committed or rolled back by that user.

A *remote transaction* contains only statements that access a single remote node. A *distributed transaction* contains statements that access more than one node.

The following sections define important concepts in transaction processing and explain how transactions access data in a distributed database:

- Remote SQL Statements
- Distributed SQL Statements
- Shared SQL for Remote and Distributed Statements
- Remote Transactions
- Distributed Transactions
- Two-Phase Commit Mechanism

- Database Link Resolution
- Schema Object Name Resolution

Remote SQL Statements

A *remote query* statement is a query that selects information from one or more remote tables, all of which reside at the same remote node. For example, the following query accesses data from the DEPT table in the SCOTT schema of the remote SALES database:

```
SELECT * FROM scott.dept@sales.us.americas.acme_auto.com;
```

A *remote update* statement is an update that modifies data in one or more tables, all of which are located at the same remote node. For example, the following query updates the DEPT table in the SCOTT schema of the remote SALES database:

```
UPDATE scott.dept@mktng.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
```

Note: A remote update can include a subquery that retrieves data from one or more remote nodes, but because the update happens at only a single remote node, the statement is classified as a remote update.

Distributed SQL Statements

A *distributed query* statement retrieves information from two or more nodes. For example, the following query accesses data from the local database as well as the remote SALES database:

```
SELECT ename, dname
  FROM scott.emp e, scott.dept@sales.us.americas.acme_auto.com d
  WHERE e.deptno = d.deptno;
```

A *distributed update* statement modifies data on two or more nodes. A distributed update is possible using a PL/SQL subprogram unit such as a procedure or trigger that includes two or more remote updates that access data on different nodes. For example, the following PL/SQL program unit updates tables on the local database and the remote SALES database:

```
BEGIN
  UPDATE scott.dept@sales.us.americas.acme_auto.com
    SET loc = 'NEW YORK'
    WHERE deptno = 10;
  UPDATE scott.emp
    SET deptno = 11
    WHERE deptno = 10;
END;
COMMIT;
```

Oracle sends statements in the program to the remote nodes, and their execution succeeds or fails as a unit.

Shared SQL for Remote and Distributed Statements

The mechanics of a remote or distributed statement using shared SQL are essentially the same as those of a local statement. The SQL text must match, and the referenced objects must match. If available, shared SQL areas can be used for the local and remote handling of any statement or decomposed query.

Remote Transactions

A *remote transaction* contains one or more remote statements, all of which reference a single remote node. For example, the following transaction contains two statements, each of which accessing the remote SALES database:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
UPDATE scott.emp@sales.us.americas.acme_auto.com
  SET deptno = 11
  WHERE deptno = 10;
COMMIT;
```

Distributed Transactions

A *distributed transaction* is a transaction that includes one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database. For example, this transaction updates the local database and the remote SALES database:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
```

```
UPDATE scott.emp
  SET deptno = 11
  WHERE deptno = 10;
COMMIT;
```

Note: If all statements of a transaction reference only a single remote node, the transaction is remote, not distributed.

Two-Phase Commit Mechanism

An database must guarantee that all statements in a transaction, distributed or non-distributed, either commit or rollback as a unit. The effects of an ongoing transaction should be invisible to all other transactions at all nodes; this transparency should be true for transactions that include any type of operation, including queries, updates, or remote procedure calls.

The general mechanisms of transaction control in a non-distributed database are discussed in the *Oracle8i Concepts*. In a distributed database, Oracle must coordinate transaction control with the same characteristics over a network and maintain data consistency, even if a network or system failure occurs.

Oracle's *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.

See Also: *Oracle8i Distributed Database Systems* for more information about Oracle's two-phase commit mechanism

Database Link Resolution

A *global object name* is an object specified using a database link. The essential components of a global object name are:

- Object name
- Database name
- Domain

The following table shows the components of an explicitly specified global database object name:

Statement	Object	Database	Domain
SELECT * FROM joan.dept@sales.acme.com	dept	sales	acme.com
SELECT * FROM emp@mktg.us.acme.com	emp	mktg	us.acme.com

Whenever a SQL statement includes a reference to a global object name, Oracle searches for a database link with a name that matches the database name specified in the global object name. For example, if you issue the following statement:

```
SELECT * FROM scott.emp@orders.us.acme.com;
```

Oracle searches for a database link called ORDERS.US.ACME.COM. Oracle performs this operation to determine the path to the specified remote database.

Oracle always searches for matching database links in the following order:

1. Private database links in the schema of the user who issued the SQL statement.
2. Public database links in the local database.
3. Global database links (only if an Oracle Name Server is available).

Resolution When the Global Database Name Is Complete

Assume you issue the following SQL statement, which specifies a complete global database name:

```
SELECT * FROM emp@prod1.us.oracle.com
```

In this case, both the database name (PROD1) and domain components (US.ORACLE.COM) are specified, so Oracle searches for private, public, and global database links. Oracle searches only for links that match the specified global database name.

Resolution When the Global Database Name Is Partial

If any part of the domain is specified, Oracle assumes that a complete global database name is specified. If a SQL statement specifies a partial global database name (that is, only the database component is specified), Oracle appends the value in the DB_DOMAIN parameter to the value in the DB_NAME parameter to construct a complete name. For example, assume you issue the following statements:

```
CONNECT scott/tiger@locdb
SELECT * FROM scott.emp@orders;
```

If the network domain for LOCDB is US.ACME.COM, then Oracle appends this domain to ORDERS to construct the complete global database name of ORDERS.US.ACME.COM. Oracle searches for database links that match only the constructed global name. If a matching link is not found, Oracle returns an error and the SQL statement cannot execute.

Resolution When No Global Database Name Is Specified

If a global object name references an object in the local database and a database link name is *not* specified using the @ symbol, Oracle automatically detects that the object is local and does not search for or use database links to resolve the object reference. For example, assume you issue the following statements:

```
CONNECT scott/tiger@locdb
SELECT * from scott.emp;
```

Because the second statement does not specify a global database name using a database link connect string, Oracle does not search for database links.

Terminating the Search for Resolution

Oracle does not necessarily stop searching for matching database links when it finds the first match. Oracle must search for matching private, public, and network database links until it determines a complete path to the remote database (both a remote account and service name).

The first match determines the remote schema as illustrated in the following table:

If you	Then Oracle	As in the example
Do *not* specify the CONNECT clause	Uses a *connected user* database link	`CREATE DATABASE LINK k1 USING 'prod'`
Do specify the CONNECT TO ... IDENTIFIED BY clause	Uses a *fixed user* database link	`CREATE DATABASE LINK k2 CONNECT TO scott IDENTIFIED BY tiger USING 'prod'`
Specify the CONNECT TO CURRENT_USER clause	Uses a *current user* database link	`CREATE DATABASE LINK k3 CONNECT TO CURRENT_USER USING 'prod'`

If you	Then Oracle	As in the example
Do *not* specify the USING clause	Searches until it finds a link specifying a database string. If matching database links are found and a string is never identified, Oracle returns an error.	`CREATE DATABASE LINK k4 CONNECT TO CURRENT_USER`

After Oracle determines a complete path, it creates a remote session—assuming that an identical connection is not already open on behalf of the same local session. If a session already exists, Oracle re-uses it.

Schema Object Name Resolution

After the local Oracle database connects to the specified remote database on behalf of the local user that issued the SQL statement, object resolution continues as if the remote user had issued the associated SQL statement. The first match determines the remote schema according to the following rules:

If you use	Then object resolution proceeds in the
A fixed user database link	Schema specified in the link creation statement.
A connected user database link	Connected user's remote schema.
A current user database link	Current user's schema.

If Oracle cannot find the object, it checks public objects of the remote database. If it cannot resolve the object, then the established remote session remains but the SQL statement cannot execute and returns an error.

Examples of Name Resolution

The following are examples of global object name resolution in a distributed database system. For all the following examples, assume that:

- The remote database is named SALES.DIVISION3.ACME.COM.
- The local database is named HQ.DIVISION3.ACME.COM.
- An Oracle Name Server (and therefore, global database links) is not available.

Example: Resolving a Complete Object Name This example illustrates how Oracle resolves a complete global object name and determines the appropriate path to the remote database using both a private and public database link. For this example, assume that a remote table EMP is contained in the schema TSMITH.

Consider the following statements issued by SCOTT at the local database:

```
CONNECT scott/tiger@hq

CREATE PUBLIC DATABASE LINK sales.division3.acme.com
CONNECT TO guest IDENTIFIED BY network
  USING 'dbstring';
```

Later, JWARD connects and issues the following statements:

```
CONNECT jward/bronco@hq

CREATE DATABASE LINK sales.division3.acme.com
  CONNECT TO tsmith IDENTIFIED BY radio;

UPDATE tsmith.emp@sales.division3.acme.com
  SET deptno = 40
  WHERE deptno = 10;
```

Oracle processes the final statement as follows:

1. Oracle determines that a complete global object name is referenced in JWARD's update statement. Therefore, the system begins searching in the local database for a database link with a matching name.
2. Oracle finds a matching private database link in the schema JWARD. Nevertheless, the private database link JWARD.SALES.DIVISION3.ACME.COM does not indicate a complete path to the remote SALES database, only a remote account. Therefore, Oracle now searches for a matching public database link.
3. Oracle finds the public database link in SCOTT's schema. From this public database link, Oracle takes the service name DBSTRING.
4. Combined with the remote account taken from the matching private fixed user database link, Oracle determines a complete path and proceeds to establish a connection to the remote SALES database as user TSMITH/RADIO.
5. The remote database can now resolve the object reference to the EMP table. Oracle searches in the TSMITH schema and finds the referenced EMP table.

6. The remote database completes the execution of the statement and returns the results to the local database.

Example: Resolving a Partial Object Name This example illustrates how Oracle resolves a partial global object name and determines the appropriate path to the remote database using both a private and public database link.

For this example, assume that:

- A table EMP on the remote database SALES is contained in the schema TSMITH, but not in schema SCOTT.
- A public synonym named EMP resides at local database HQ and points to TSMITH.EMP at the remote database SALES.
- The public database link in "Example: Resolving a Complete Object Name" on page 30-40 is already created on local database HQ:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com
  CONNECT TO guest IDENTIFIED BY network
  USING 'dbstring';
```

Consider the following statements issued at local database HQ:

```
CONNECT scott/tiger@hq

CREATE DATABASE LINK sales.division3.acme.com;

DELETE FROM emp@sales
  WHERE empno = 4299;
```

Oracle processes the final DELETE statement as follows:

1. Oracle notices that a partial global object name is referenced in SCOTT's DELETE statement. It expands it to a complete global object name using the domain of the local database as follows:

   ```
   DELETE FROM emp@sales.division3.acme.com
     WHERE empno = 4299;
   ```

2. Oracle searches the local database for a database link with a matching name.
3. Oracle finds a matching *private* connected user link in the schema SCOTT, but the private database link indicates no path at all. Oracle uses the connected username/password as the remote account portion of the path and then searches for and finds a matching *public* database link:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com
  CONNECT TO guest IDENTIFIED BY network
  USING 'dbstring';
```

4. Oracle takes the database net service name DBSTRING from the public database link. At this point, Oracle has determined a complete path.
5. Oracle connects to the remote database as SCOTT/TIGER and searches for and does not find an object named EMP in the schema SCOTT.
6. The remote database searches for a public synonym named EMP and finds it.
7. The remote database executes the statement and returns the results to the local database.

Global Name Resolution in Views, Synonyms, and Procedures

A view, synonym, or PL/SQL program unit (e.g., procedure, function, or trigger) can reference a remote schema object by its global object name. If the global object name is complete, then Oracle stores the definition of the object without expanding the global object name. If the name is partial, however, Oracle expands the name using the domain of the local database name.

The following table explains when Oracle completes the expansion of a partial global object name for views, synonyms, and program units:

If you	Then Oracle
Create a view	Does *not* expand partial global names—the data dictionary stores the exact text of the defining query. Instead, Oracle expands a partial global object name each time a statement that uses the view is parsed.
Create a synonym	Expands partial global names. The definition of the synonym stored in the data dictionary includes the expanded global object name.
Compile a program unit	Expands partial global names.

What Happens When Global Names Change

Global name changes can affect views, synonyms, and procedures that reference remote data using partial global object names. If the global name of the referenced database changes, views and procedures may try to reference a nonexistent or incorrect database. On the other hand, synonyms do not expand database link names at runtime, so they do not change.

For example, consider two databases named SALES.UK.ACME.COM and HR.UK.ACME.COM. Also, assume that the SALES database contains the following view and synonym:

```
CREATE VIEW employee_names AS
        SELECT ename FROM scott.emp@hr;

CREATE SYNONYM employee FOR scott.emp@hr;
```

Oracle expands the EMPLOYEE synonym definition and stores it as:

```
scott.emp@hr.uk.acme.com
```

Scenario 1: Both Databases Change Names First, consider the situation where both the Sales and Human Resources departments are relocated to the United States. Consequently, the corresponding global database names are both changed as follows:

- SALES.US.ACME.COM
- HR.US.ACME.COM

Query on SALES	Expansion Before Change	Expansion After Change
SELECT * FROM employee_names	SELECT * from scott.emp@hr.**uk**.acme.com	SELECT * from scott.emp@hr.**us**.acme.com
SELECT * FROM employee	SELECT * FROM scott.emp@hr.**uk**.acme.com	SELECT * FROM scott.emp@hr.**uk**.acme.com

Scenario 2: One Database Changes Names Now consider that only the Sales department is moved to the United States; Human Resources remains in the UK. Consequently, the global database names are now:

- SALES.US.ACME.COM
- HR.UK.ACME.COM

Query on SALES	Expansion Before Change	Expansion After Change
SELECT * FROM employee_names	SELECT * FROM scott.emp@hr.**uk**.acme.com	SELECT * FROM scott.emp@hr.**us**.acme.com
SELECT * FROM employee	SELECT * FROM scott.emp@hr.**uk**.acme.com	SELECT * FROM scott.emp@hr.**uk**.acme.com

In this case, the defining query of the EMPLOYEE_NAMES view expands to a non-existent global database name. On the other hand, the EMPLOYEE synonym continues to reference the correct database, HR.UK.ACME.COM.

Distributed Database Application Development

Application development in a distributed system raises issues that are not applicable in a non-distributed system. This section contains the following topics relevant for distributed application development:

- Transparency in a Distributed Database System
- Remote Procedure Calls
- Distributed Query Optimization

> **See Also:** *Oracle8i Distributed Database Systems* to learn how to develop applications for distributed systems

Transparency in a Distributed Database System

With minimal effort, you can develop application that make an Oracle distributed database system transparent to users that work with the system. The goal of transparency is to make a distributed database system appear as though it is a single Oracle database. Consequently, the system does not burden developers and users of the system with complexities that would otherwise make distributed database application development challenging and detract from user productivity.

The following sections explain more about transparency in a distributed database system.

Location Transparency

An Oracle distributed database system has features that allow application developers and administrators to hide the physical location of database objects from applications and users. *Location transparency* exists when a user can universally refer to a database object such as a table, regardless of the node to which an application connects. Location transparency has several benefits, including:

- Access to remote data is simple, because database users do not need to know the physical location of database objects.
- Administrators can move database objects with no impact on end-users or existing database applications.

Most typically, administrators and developers use synonyms to establish location transparency for the tables and supporting objects in an application schema. For example, the following statements create synonyms in a database for tables in another, remote database.

```
CREATE PUBLIC SYNONYM emp
  FOR scott.emp@sales.us.americas.acme_auto.com
CREATE PUBLIC SYNONYM dept
  FOR scott.dept@sales.us.americas.acme_auto.com
```

Now, rather than access the remote tables with a query such as:

```
SELECT ename, dname
  FROM scott.emp@sales.us.americas.acme_auto.com e,
       scott.dept@sales.us.americas.acme_auto.com d
  WHERE e.deptno = d.deptno;
```

an application can issue a much simpler query that does not have to account for the location of the remote tables.

```
SELECT ename, dname
  FROM emp e, dept d
  WHERE e.deptno = d.deptno;
```

In addition to synonyms, developers can also use views and stored procedures to establish location transparency for applications that work in a distributed database system.

SQL and COMMIT Transparency

Oracle's distributed database architecture also provides query, update, and transaction transparency. For example, standard SQL statements such as SELECT, INSERT, UPDATE, and DELETE work just as they do in a non-distributed database environment. Additionally, applications control transactions using the standard SQL statements COMMIT, SAVEPOINT, and ROLLBACK—there is no requirement for complex programming or other special operations to provide distributed transaction control.

- The statements in a single transaction can reference any number of local or remote tables.
- Oracle guarantees that all nodes involved in a distributed transaction take the same action: they either all commit or all roll back the transaction.
- If a network or system failure occurs during the commit of a distributed transaction, the transaction is automatically and transparently resolved

globally; that is, when the network or system is restored, the nodes either all commit or all roll back the transaction.

Internal Operations Each committed transaction has an associated *system change number* (*SCN*) to uniquely identify the changes made by the statements within that transaction. In a distributed database, the SCNs of communicating nodes are coordinated when:

- A connection is established using the path described by one or more database links
- A distributed SQL statement is executed
- A distributed transaction is committed

Among other benefits, the coordination of SCNs among the nodes of a distributed database system allows global distributed read-consistency at both the statement and transaction level. If necessary, global distributed time-based recovery can also be completed.

Replication Transparency

Oracle also provide many features to transparently replicate data among the nodes of the system. For more information about Oracle's replication features, see *Oracle8i Replication*.

Remote Procedure Calls

Developers can code PL/SQL packages and procedures to support applications that work with a distributed database. Applications can make local procedure calls to perform work at the local database and *remote procedure calls (RPCs)* to perform work at a remote database.

When a program calls a remote procedure, the local server passes all procedure parameters to the remote server in the call. For example, the following PL/SQL program unit calls the packaged procedure DEL_EMP located at the remote SALES database and passes it the parameter 1257:

```
BEGIN
 emp_mgmt.del_emp@sales.us.americas.acme_auto.com(1257);
END;
```

In order for the RPC to succeed, the called procedure must exist at the remote site.

When developing packages and procedures for distributed database systems, developers must code with an understanding of what program units should do at remote locations, and how to return the results to a calling application.

Distributed Query Optimization

Distributed query optimization is a default Oracle8*i* feature that reduces the amount of data transfer required between sites when a transaction retrieves data from remote tables referenced in a distributed SQL statement.

Distributed query optimization uses Oracle's cost-based optimization to find or generate SQL expressions that extract only the necessary data from remote tables, process that data at a remote site or sometimes at the local site, and send the results to the local site for final processing. This operation reduces the amount of required data transfer when compared to the time it takes to transfer all the table data to the local site for processing.

Using various cost-based optimizer hints such as DRIVING_SITE, NO_MERGE, and INDEX, you can control where Oracle processes the data and how it accesses the data.

> **See Also:** *Oracle8i Distributed Database Systems* for more information about cost-based optimization

National Language Support

Oracle supports environments in which clients, Oracle servers, and non-Oracle servers use different character sets. In Oracle8*i*, NCHAR support is provided for heterogeneous environments. You can set a variety of NLS and HS parameters to control data conversion between different character sets.

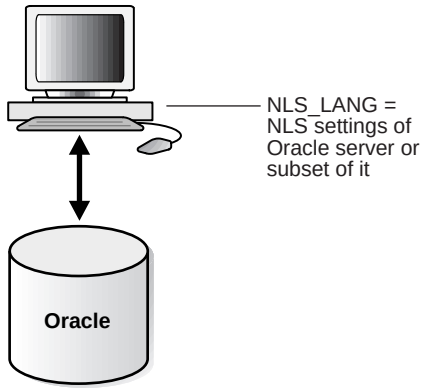
Character settings are defined by the following NLS and HS parameters:

Parameters	Environment	Defined For
NLS_LANG	Client-Server	Client
NLS_LANGUAGE NLS_CHARACTERSET NLS_TERRITORY	Client-Server Non-Heterogeneous Distributed Heterogeneous Distributed	Oracle server
HS_LANGUAGE	Heterogeneous Distributed	Non-Oracle server Transparent gateway
NLS_NCHAR HS_NLS_NCHAR	Heterogeneous Distributed	Oracle server Transparent gateway

Client-Server Environment

In a client-server environment, set the client character set to be the same as or a subset of the Oracle server character set, as illustrated in Figure 30–6:

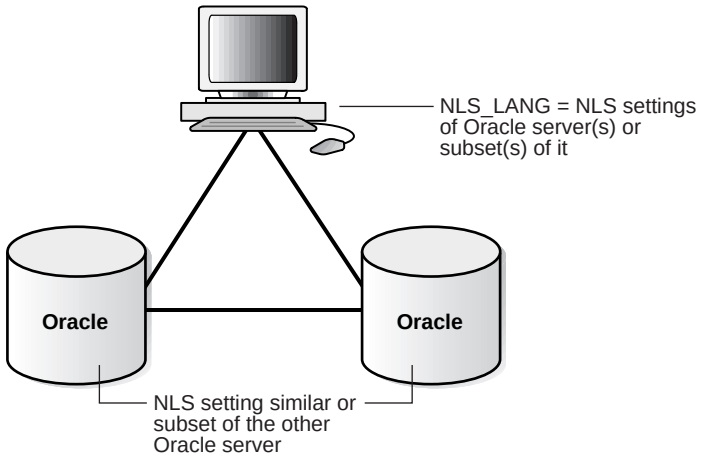
Figure 30–6 NLS Settings in a Client-Server Environment



Homogeneous Distributed Environment

In a non-heterogeneous environment, the client and server character sets should be either the same as or subsets of the main server character set, as illustrated in Figure 30–7:

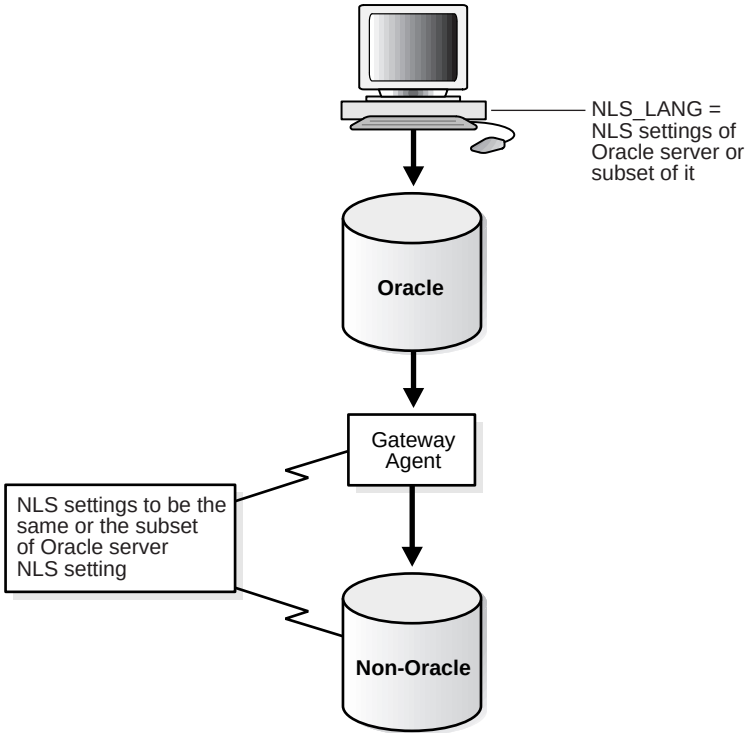
Figure 30–7 NLS Settings in a Homogeneous Environment



Heterogeneous Distributed Environment

In a heterogeneous environment, the NLS settings of the client, the transparent gateway and the non-Oracle data source, should be either the same or a subset of the Oracle server NLS character set as illustrated in Figure 30–8. Transparent gateways have full NLS support.

Figure 30–8 NLS Settings in a Heterogeneous Environment



In a heterogeneous environment, only transparent gateways built with HS technology support complete NCHAR capabilities. Whether a specific transparent gateway supports NCHAR depends on the non-Oracle data source it is targeting. For further information on how a particular transparent gateway handles NCHAR support, consult the particular transparent gateway documentation.

> **See Also:** *Oracle8i Reference* for more information about National Language Support features

31

Replication

This chapter explains the basic concepts and terminology behind Oracle replication. This chapter covers the following topics:

- Introduction to Replication
- Applications That Use Replication
- Replication Objects, Groups, and Sites
- Types of Replication Environments
- Administration Tools for a Replication Environment
- Replication Conflicts
- Other Options for Multimaster Replication

Note: If you are using Trusted Oracle, see your documentation for Oracle security-related products for information about using replication in that environment.

Introduction to 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 might 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 might be available at DB1, DB2, and DB3.

Some of the common reasons for using replication are:

Availability Replication improves the availability of applications because it provides them with alternative data access options. If one site becomes unavailable, users can continue to query or even update the remaining locations. In other words, replication provides excellent failover protection.

Performance Replication provides fast, local access to shared data because it balances activity over multiple sites. Some users can access one server while other users access other servers, thereby reducing the load at all servers. Also, users can access data from the replication site that has the lowest access cost, which is typically the site that is geographically closest to them.

Disconnected Computing A *snapshot* is a complete or partial copy (replica) of a target master table from a single point in time. Snapshots enable users to work on a subset of a database while disconnected from the central database server. Later, when a connection is established, users can synchronize (refresh) snapshots on demand. When users refresh snapshots, they update the central database with all of their changes, and they receive any changes that may have happened while they were disconnected.

Network Load Reduction Replication can be used to distribute data over multiple regional locations. Then, applications can access various regional servers instead of accessing one central server. This configuration can reduce network load dramatically.

Mass Deployment Increasingly, organizations need to deploy many applications that require the ability to use and manipulate data. With Oracle replication, deployment templates enable you to create multiple snapshot environments quickly. You can use variables to customize each snapshot environment for its individual needs. For example, you can use deployment templates for sales force automation; in this case, the template could contain variables for various sales regions and salespersons.

You will find more detailed descriptions of the uses of replication in later chapters.

Applications That Use Replication

Replication supports a variety of applications that often have different requirements. Some applications allow for relatively autonomous individual snapshot sites. For example, sales force automation, field service, retail, and other mass deployment applications typically require data to be periodically synchronized between central database systems and a large number of small, remote sites, which are often disconnected from the central database. Members of a sales force must be able to complete transactions, regardless of whether they are connected to the central database. In this case, remote sites must be autonomous.

On the other hand, applications such as call centers and Internet systems require data on multiple servers to be synchronized in a continuous, nearly instantaneous manner to ensure that the service provided is available and equivalent at all times. For example, a retail web site on the Internet must ensure that customers see the same information in the online catalog at each site. Here, data consistency is more important than site autonomy.

Oracle replication can be used for both types of applications, and for applications that combine aspects of both. In fact, one Oracle replication environment can support both mass deployment and server-to-server replication, which enables integration into one coherent environment. In such an environment, for example, sales force automation and customer service call centers can share data.

Replication Objects, Groups, and Sites

The following sections explain the basic components of a replication system, including replication objects, replication groups, and replication sites.

Replication Objects

A *replication object* is a database object existing on multiple servers in a distributed database system. In a replication environment, any updates made to a replication object at one site are applied to the copies at all other sites. Oracle replication enables you to replicate the following types of objects:

- Tables
- Indexes
- Views
- Packages and Package Bodies
- Procedures and Functions
- Triggers
- Sequences
- Synonyms

Replication Groups

In a replication environment, Oracle manages replication objects using *replication groups*. A replication group is a collection of replication objects that are logically related. The objects in a replication group are administered together.

By organizing related database objects within a replication group, it is easier to administer many objects together. Typically, you create and use a replication group to organize the schema objects necessary to support a particular database application. However, replication groups and schemas do not need to correspond with one another. A replication group can contain objects from multiple schemas, and a single schema can have objects in multiple replication groups. However, a replication object can be a member of only one replication group.

Replication Sites

A replication group can exist at multiple *replication sites*. Replication environments support two basic types of sites: *master sites* and *snapshot sites*. One site can be both a master site and a snapshot site at the same time.

The differences between master sites and snapshot sites are:

- A replication group at a master site is more specifically referred to as a *master group*. A replication group at a snapshot site is more specifically referred to as a *snapshot group*. Additionally, every master group has exactly one *master definition site*. A replication group's master definition site is a master site serving as the control center for managing the replication group and the objects in the group.
- A master site maintains a complete copy of all objects in a replication group, while snapshots at a snapshot site can contain all or a subset of the table data within a master group. For example, if the SCOTT_MG master group contains the tables EMP and DEPT, all of the master sites must maintain a complete copy of EMP and DEPT. However, one snapshot site might contain only a snapshot of the EMP table, while another snapshot site might contain snapshots of both the EMP and DEPT tables.
- All master sites in a multimaster replication environment communicate directly with one another to continually propagate data and schema changes in the replication group. Snapshot sites contain an image, or snapshot, of the table data from a certain point in time. Typically, a snapshot is refreshed periodically to synchronize it with its master site. You can organize snapshots into *refresh groups*. Snapshots in a refresh group can belong to one or more snapshot groups, and they are refreshed at the same time to ensure that the data of all snapshots in the refresh group correspond to the same transactionally consistent point in time.

Types of Replication Environments

Oracle replication supports the following types of replication environments:

- Multimaster Replication
- Snapshot Replication
- Multimaster and Snapshot Hybrid Configurations

Multimaster Replication

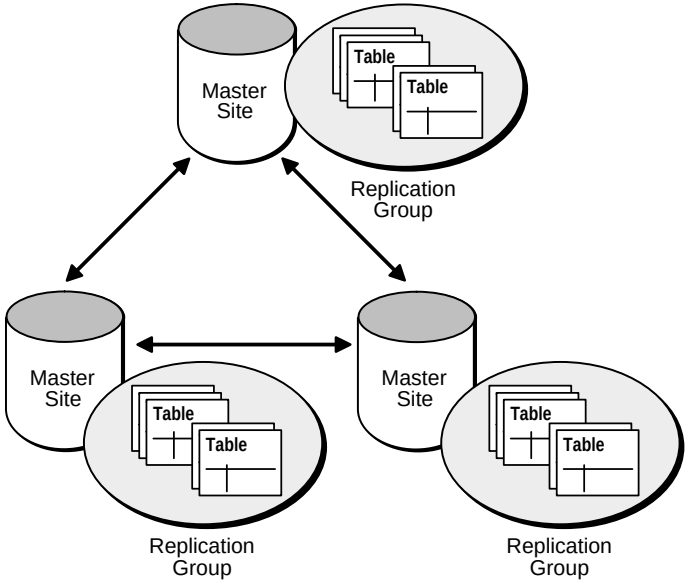
Multimaster replication (also called peer-to-peer or *n*-way replication) allows multiple sites, acting as equal peers, to manage groups of replicated database objects. Each site in a multimaster replication environment is a master site.

Applications can update any replicated table at any site in a multimaster configuration. Oracle database servers operating as master sites in a multimaster environment automatically work to converge the data of all table replicas and to ensure global transaction consistency and data integrity.

Asynchronous replication is the most common way to implement multimaster replication. Other ways include synchronous replication and procedural replication, which are discussed later in this chapter. When you use asynchronous replication, an update of a table is stored in the deferred transactions queue at the master site where the change occurred. These changes are called *deferred transactions*. The deferred transactions are pushed (or propagated) to the other participating master sites at regular intervals. You can control the amount of time in an interval.

Using asynchronous replication means that conflicts are possible because the same row value might be updated at two different master sites at nearly the same time. However, you can use techniques to avoid conflicts and, if conflicts occur, Oracle provides built-in mechanisms to resolve them.

Figure 31–1 Multimaster Replication



Quiescing Master Groups

At times, you must stop all replication activity for a master group so that you can perform certain administrative tasks on the master group. For example, you must stop all replication activity for a master group to issue data definition language (DDL) statements on any table in the group. Stopping all replication activity for a master group is called *quiescing* the group. When a master group is quiesced, users cannot perform data manipulation language (DML) statements on any of the objects in the master group.

Snapshot Replication

A snapshot contains a complete or partial copy of a target master table from a single point in time. A snapshot may be read-only or updateable.

All snapshots provide the following benefits:

- Enable local access, which improves response times and availability.
- Offload queries from the master site, because users can query the local snapshot instead.
- Increase data security by allowing you to replicate only a selected subset of the target master table's data set.

Read-Only Snapshots

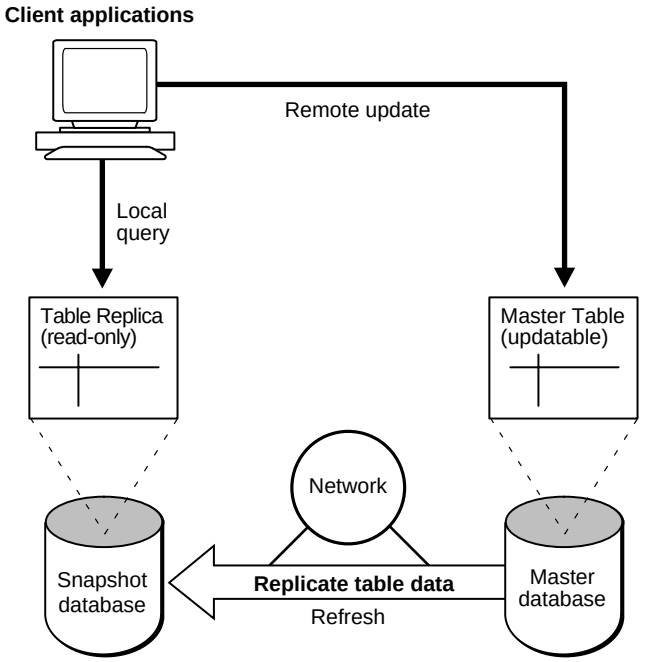
In a basic configuration, snapshots can provide read-only access to the table data that originates from a master site. Applications can query data from read-only snapshots to avoid network access regardless of network availability. However, applications throughout the system must access data at the master site to perform an update. Figure 31–2 illustrates basic, read-only replication. The master tables of read-only snapshots do not need to belong to a master group.

Read-only snapshots provide the following benefits:

- Eliminate the possibility of conflicts because they cannot be updated.
- Support complex snapshots. Examples of complex snapshots are snapshots that contain set operations or a CONNECT BY clause.

> **See Also:** *Oracle8i Replication* for more information about complex snapshots

Figure 31–2 Read-Only Snapshot Replication

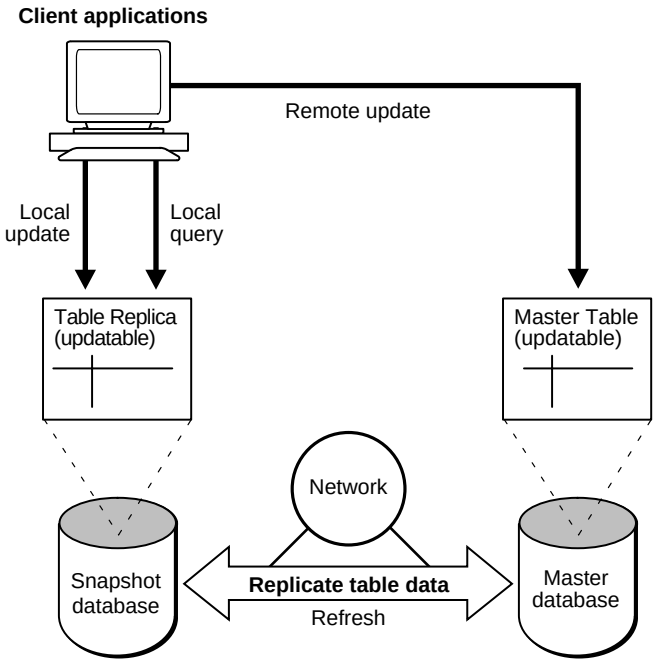


Updatable Snapshots

In a more advanced configuration, you can create an *updatable snapshot* that allows users to insert, update, and delete rows of the target master table by performing these operations on the snapshot. An updatable snapshot may also contain only a subset of the data in the target master table. Figure 31–3 illustrates a replication environment using updatable snapshots.

Updatable snapshots are based on tables at a master site that have been set up to support replication. In fact, updatable snapshots must be part of a snapshot group that is based on a master group at a master site.

Figure 31–3 Updatable Snapshot Replication



Updatable snapshots have the following properties.

- Updatable snapshots are always based on a single table.
- Updatable snapshots can be incrementally (or "fast") refreshed.
- Oracle propagates the changes made to an updatable snapshot to the snapshot's remote master table. If necessary, the updates to the master table then cascade to all other master sites.
- Oracle can refresh an updatable snapshot as part of a refresh group in the same way it refreshes read-only snapshots.

Updatable snapshots provide the following benefits:

- Allow users to query and update a local replicated data set even when disconnected from the master site.
- Require fewer resources than multimaster replication, while still supporting data updates. For example, because snapshots can reside on an Oracle8*i* Lite database, the disk space and memory requirements for snapshot clients are much less than the requirements for an Oracle8*i* server.

Snapshot Refresh

To ensure that a snapshot is consistent with its master table, you need to *refresh* the snapshot periodically. Oracle provides the following three methods to refresh snapshots:

- *Fast refresh* uses snapshot logs to update only the rows that have changed since the last refresh.
- *Complete refresh* updates the entire snapshot.
- *Force refresh* performs a fast refresh when possible. When a fast refresh is not possible, force refresh performs a complete refresh.

When it is important for snapshots to be transactionally consistent with each other, you can organize them into *refresh groups*. By refreshing the refresh group, you can ensure that the data in all of the snapshots in the refresh group correspond to the same transactionally consistent point in time. A snapshot in a refresh group still can be refreshed individually, but doing so nullifies the benefits of the refresh group because refreshing the snapshot individually does not refresh the other snapshots in the refresh group.

Snapshot Log

A *snapshot log* is a table that records all of the DML changes to a master table. A snapshot log is associated with a single master table, and each master table has only one snapshot log, regardless of how many snapshots refresh from the master. A fast refresh of a snapshot is possible only if the snapshot's master table has a snapshot log. When a snapshot is fast refreshed, entries in the snapshot's associated snapshot log that have appeared since the snapshot was last refreshed are applied to the snapshot.

Deployment Templates

Deployment templates simplify the task of deploying and maintaining many remote snapshot sites. Using deployment templates, you can define a collection of snapshot definitions at a master site, and you can use parameters in the definitions so that the snapshots can be customized for individual users or types of users.

For example, you might create one template for the sales force and another template for field service representatives. In this case, a parameter value might be the sales territory or the customer support level. When a remote user connects to a master site, the user sees a list of available templates. When the user *instantiates* a template, the appropriate snapshots are created and populated at the remote site. The appropriate parameter values can either be supplied by the remote user or taken from a table maintained at the master site.

Online and Offline Instantiation When a user instantiates a template at a snapshot site, the object DDL (for example, CREATE SNAPSHOT... or CREATE TABLE...) is executed to create the appropriate schema objects at the snapshot site, and the objects are populated with the appropriate data.

Users can instantiate templates while connected to the master site over a network (online instantiation), or while disconnected from the master site (offline instantiation).

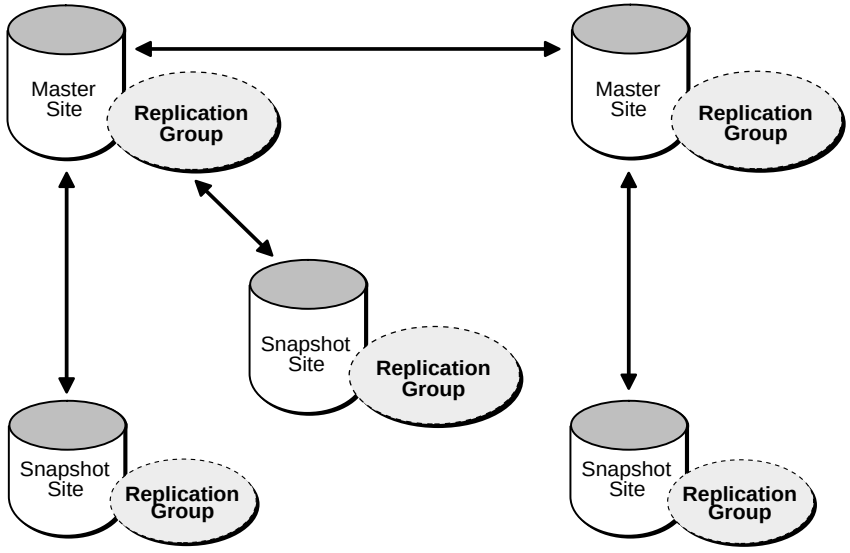
Offline instantiation is often used to decrease server loads during peak usage periods and to reduce remote connection times. To instantiate a template offline, you package the template and required data on some type of storage media, such as tape, CD-ROM, and so on. Then, instead of pulling the data from the master site, users pull the data from the storage media containing the template and data.

Multimaster and Snapshot Hybrid Configurations

Multimaster replication and snapshots can be combined in *hybrid* or "mixed" configurations to meet different application requirements. Mixed configurations can have any number of master sites and multiple snapshot sites for each master.

For example, as shown in Figure 31–4, multimaster (or *n*-way) replication between two masters can support full-table replication between the databases that support two geographic regions. Snapshots can be defined on the masters to replicate full tables or table subsets to sites within each region.

Figure 31–4 Hybrid Configuration



Key differences between snapshots and replicated masters include the following:

- Replicated masters must contain data for the full table being replicated, whereas snapshots can replicate subsets of master table data.
- Multimaster replication allows you to replicate changes for each transaction as the changes occur. Snapshot refreshes are set oriented, propagating changes from multiple transactions in a more efficient, batch-oriented operation, but at less frequent intervals.
- Master sites detect and resolve the conflicts that occur from changes made to multiple copies of the same data.

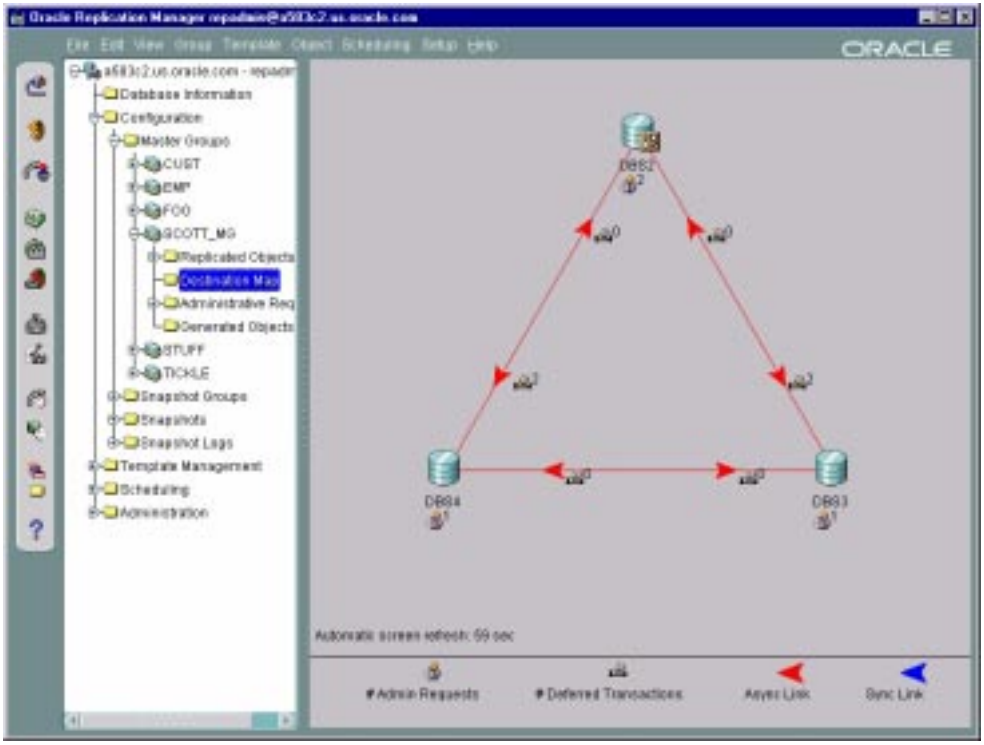
Administration Tools for a Replication Environment

Several tools are available for administering and monitoring your replication environment. Oracle's Replication Manager provides a powerful GUI interface to help you manage your environment, while the replication management API provides you with the familiar application programming interface (API) to build customized scripts for replication administration. Additionally, the replication catalog keeps you informed about your replicated environment.

Oracle Replication Manager

Replication environments supporting both multimaster and snapshot replication can be challenging to configure and manage. To help administer these replication environments, Oracle provides a sophisticated management tool called Oracle Replication Manager. Other sections in this book include information and examples for using Replication Manager. However, the Replication Manager online help is the primary documentation source for Replication Manager.

Figure 31–5 Replication Manager



See Also: *Oracle8i Replication* for an introduction to Replication Manager, and see the Replication Manager online help for complete instructions on using Replication Manager.

Replication Management API

The replication management application programming interface (API) is a set of PL/SQL packages that encapsulate procedures and functions that you can use to configure an Oracle replication environment. The replication management API is a command-line alternative to Replication Manager. In fact, Replication Manager uses the procedures and functions of the replication management API to perform its work. For example, when you use Replication Manager to create a new master group, Replication Manager completes the task by making a call to the DBMS_REPCAT.CREATE_MASTER_REPGROUP procedure. The replication management API makes it easy for you to create custom scripts to manage your replication environment.

> **See Also:** *Oracle8i Replication Management API Reference* for more information about using the replication management API.

Replication Catalog

Every master and snapshot site in a replication environment has a *replication catalog*. A replication catalog for a site is a distinct set of data dictionary tables and views that maintain administrative information about replication objects and replication groups at the site. Every server participating in a replication environment can automate the replication of objects in replication groups using the information in its replication catalog.

> **See Also:** *Oracle8i Replication Management API Reference* for more information about the replication catalog.

Distributed Schema Management

In a replication environment, all DDL statements must be issued using either Replication Manager or the DBMS_REPCAT package. When you use either of these interfaces, all DDL statements are replicated to all of the sites participating in the replication environment.

Note: Any DDL statements issued directly using a tool such as SQL*Plus are not replicated to other sites.

Replication Conflicts

Asynchronous multimaster and updateable snapshot replication environments must address the possibility of replication conflicts that may occur when, for example, two transactions originating from different sites update the same row at nearly the same time. When data conflicts occur, you need a mechanism to ensure that the conflict is resolved in accordance with your business rules and to ensure that the data converges correctly at all sites.

In addition to logging any conflicts that may occur in your replicated environment, Oracle replication offers a variety of built-in conflict resolution methods that enable you to define a conflict resolution system for your database that resolves conflicts in accordance with your business rules. If you have a unique situation that Oracle's built-in conflict resolution methods cannot resolve, you have the option of building and using your own conflict routines.

See Also:

- *Oracle8i Replication* for information about how to design your database to avoid data conflicts and how to build conflict resolution routines that resolve such conflicts when they occur
- The online help for Replication Manager for instructions on using Replication Manager to configure conflict resolution methods
- *Oracle8i Replication Management API Reference* for a description of how to build conflict resolution routines using the replication management API

Other Options for Multimaster Replication

Asynchronous replication is the most common way to implement multimaster replication. However, you have two other options: synchronous replication and procedural replication.

Synchronous Replication

A multimaster replication environment can use either asynchronous or synchronous replication to copy data. With asynchronous replication, changes made at one master site occur at a later time at all other participating master sites. With synchronous replication, changes made at one master site occur immediately at all other participating master sites.

When you use synchronous replication, an update of a table results in the immediate replication of the update at all participating master sites. In fact, each transaction includes all master sites. Therefore, if one master site cannot process a transaction for any reason, the transaction is rolled back at all master sites.

Although you avoid the possibility of conflicts when you use synchronous replication, it requires a very stable environment to operate smoothly. If communication to one master site is not possible because of a network problem, for example, no transactions can be completed until communication is re-established.

Procedural Replication

Batch processing applications can change large amounts of data within a single transaction. In such cases, typical row-level replication might load a network with many data changes. To avoid such problems, a batch processing application operating in a replication environment can use Oracle's *procedural replication* to replicate simple stored procedure calls to converge data replicas. Procedural replication replicates only the call to a stored procedure that an application uses to update a table. It does not replicate the data modifications themselves.

To use procedural replication, you must replicate the packages that modify data in the system to all sites. After replicating a package, you must generate a *wrapper* for the package at each site. When an application calls a packaged procedure at the local site to modify data, the wrapper ensures that the call is ultimately made to the same packaged procedure at all other sites in the replicated environment. Procedural replication can occur asynchronously or synchronously.

Conflict Detection and Procedural Replication

When a replication system replicates data using procedural replication, the procedures that replicate data are responsible for ensuring the integrity of the replicated data. That is, you must design such procedures to either avoid or detect replication conflicts and to resolve them appropriately. Consequently, procedural replication is most typically used when databases are modified only with large batch operations. In such situations, replication conflicts are unlikely because numerous transactions are not contending for the same data.

See Also: *Oracle8i Replication*

Part X

Appendix

Part X contains the following appendix:

- Appendix A, "Operating System-Specific Information"

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 may vary on different operating systems. Throughout this manual, references to these manuals are marked with the icon shown in the left margin.

This appendix lists all the references in this manual to operating system-specific Oracle manuals, and lists the operating system (OS) 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 below.

- Administrator privileges, prerequisites: "Connecting with Administrator Privileges" on page 5-3
- Auditing: "Events Always Audited to the Operating System Audit Trail" on page 28-5 and "Auditing to the Operating System Audit Trail" on page 28-6
- Authenticating users: "Authentication by the Operating System" on page 26-4
- Authenticating DBAs: "Connecting with Administrator Privileges" on page 5-3 and "Authentication of Database Administrators" on page 26-13
- Background processes, ARC*n*: "Archiver Processes (ARCn)" on page 8-12
- Background processes, creating: "Background Processes" on page 8-5
- Background processes, DBW*n* processes: "Database Writer (DBWn)" on page 8-8
- Client / server communication: "Dedicated Server Configuration" on page 8-23
- Communication software: "Operating System Communications Software" on page 8-28
- Configuring Oracle: "Types of Processes" on page 8-2
- Data blocks, size of: "Data Blocks" on page 4-3
- Datafiles, size of file header: "Datafiles" on page 3-16
- Dedicated server, requesting for administrative operations: "Restricted Operations of the Multi-Threaded Server" on page 8-21
- Indexes, overhead of index blocks: "Format of Index Blocks" on page 10-29
- Net8, choosing and installing network drivers: "The Program Interface Drivers" on page 8-28
- Net8, drivers included in Net8 software: "How Net8 Works" on page 6-5
- Parallel recovery and asynchronous I / O: "Situations That Benefit from Parallel Recovery" on page 29-11
- Program global areas (PGAs): "Size of a PGA" on page 7-16
- Role management by the operating system: "The Operating System and Roles" on page 27-23
- Rollback segments, number of transactions per: "Transactions and Rollback Segments" on page 4-22
- Software code areas, shared or unshared: "Software Code Areas" on page 7-18

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