Oracle7™ Server Concepts

Release 7.3

February 1996

Part No. A32534–1

ORACLE®

Oracle7™ Server Concepts, Release 7.3

Part No. A32534–1

Copyright © 1993, 1996 Oracle Corporation

All rights reserved. Printed in the U.S.A.

Contributing Authors: Steven Bobrowski, Cynthia Chin–Lee, Cindy Closkey, John Frazzini, Danny Sokolsky

Technical Illustrator: Valarie Moore

Contributors: Richard Allen, David Anderson, Andre Bakker, Bill Bridge, Atif Chaudry, Jeff Cohen, Sandy Dreskin, Jason Durbin, Ahmed Ezzat, Anurag Gupta, Gary Hallmark, Michael Hartstein, Terry Hayes, Ken Jacobs, Sandeep Jain, Amit Jasuja, Hakan Jakobsson, Robert Jenkins, Jr., Jonathan Klein, R. Kleinro, Robert Kooi, Juan Loaiza, Brom Mahbod, William Maimone, Andrew Mendelsohn, Mark Moore, Mark Porter, Maria Pratt, Tuomas Pystenen, Patrick Ritto, Hasan Rizvi, Hari Sankar, Gordon Smith, Leng Leng Tan, Lynne Thieme, Alex Tsukerman, Joyo Wijaya, Linda Willis

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer’s responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227–7013, Rights in Technical Data and Computer Software (October 1988).

Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with “Restricted Rights”, as defined in FAR 52.227–14, Rights in Data – General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

Oracle, Pro*Ada, Pro*COBOL, Pro*FORTRAN, Pro*Pascal, Pro*PL/I, SQL*DBA,SQL*Loader, SQL*Net, SQL*Plus, and SQL*ReportWriter are registered trademarks of Oracle Corporation.

Oracle7, Oracle Forms, Oracle Parallel Server, PL/SQL, and Pro*C are trademarks of Oracle Corporation.

VMS is a registered trademark of Digital Equipment Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Preface

This manual describes all features of the Oracle7 Server, a relational database management system (RDBMS). This manual describes how how Oracle functions. It lays a conceptual foundation for much of the practical information contained in other Oracle7 Server manuals.

Information in this manual applies to the Oracle7 Server running on all operating systems. It provides information about the base Oracle Server product and the special options, including the:

- distributed option
- advanced replication option
- parallel query option
- Parallel Server option

Any chapter of this manual that applies to a particular option only is indicated on its first page.

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 they are running Oracle.

As a prerequisite, **all readers should read the first chapter**, "Introduction to the Oracle Server". Chapter 1 is a comprehensive introduction to the concepts and terminology used throughout the remainder of this manual.

If You're 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 *Oracle7 Server Migration*.

If You're 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 the *Oracle7 Server Administrator's Guide.*

If You're 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 the *Oracle7 Server Application Developer's Guide* and to the documentation for the tool or language product they are using to develop Oracle database applications.

How *Oracle7 Server Concepts* Is Organized

This manual is divided into the chapters described below.

Part I
What is Oracle?

Chapter 1 Introduction to the Oracle Server

This chapter provides the "big picture", outlining the concepts and terminology you need to understand the Oracle Server. You should read this overview before using the detailed information in the remainder of this manual.

Part II
Basic Database Operation

Chapter 2 Database and Instance Startup and Shutdown

This chapter describes how the database administrator (DBA) can control the accessibility of an Oracle database system. This chapter also describes the parameters that control how the database operates.

Part III
Database Structures

Chapter 3 Data Blocks, Extents, and Segments

This chapter discusses how data is stored and how storage space is allocated for and consumed by various objects within an Oracle database. The space management background information here supplements that in the following two chapters.

Chapter 4 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 5 Schema Objects

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

Chapter 6 Datatypes

This chapter describes the types of 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). Among the issues covered are the following:

- how data of different types is physically stored
- datatype size limits
- conversion of Oracle datatypes from one to another
- how datatypes of other systems are mapped to Oracle datatypes

Chapter 7 Data Integrity

This chapter discusses data integrity and the declarative integrity constraints used to enforce it.

Chapter 8 The Data Dictionary

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

Part IV System Architecture

Chapter 9 Memory Structures and Processes

This chapter describes the memory structures and processes that make up an Oracle database system. This chapter also describes the different process configurations available for Oracle.

Chapter 10 Data Concurrency

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

Part V Data Access

Chapter 11 SQL and PL/SQL

This chapter briefly describes SQL (the Structured Query Language), the language used to communicate with Oracle, as well as PL/SQL, Oracle's procedural language extension to SQL.

Chapter 12 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 13 The Optimizer

This chapter explains how the optimizer works. The optimizer is the part of Oracle that chooses the most efficient way to execute each SQL statement.

Part VI Programmatic Constructs

Chapter 14 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 15 Database Triggers

This chapter describes the procedural language constructs called triggers, procedures that are implicitly executed when anyone inserts rows into, updates, or deletes rows from a database table.

Chapter 16 Dependencies Among Schema Objects

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

Part VII Database Security

Chapter 17 Database Access

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

Chapter 18 Privileges and Roles

This chapter discusses system and object security.

Chapter 19 Auditing

This chapter discusses how Oracle's auditing feature tracks database activity.

Part VIII Distributed Processing and Distributed Databases

Chapter 20 Client/Server Architecture

This chapter discusses distributed processing environments and the Oracle Server.

Chapter 21 Distributed Databases

This chapter discusses the Oracle Server's distributed architecture, remote data access, and table replication.

Part IX
Database Backup and Recovery

Chapter 22 Recovery Structures

This chapter describes the files and structures used for database recovery: the redo log files and the control files. Media and software failure are covered.

Chapter 23 Database Backup

This chapter discusses how to protect an Oracle database from possible failures.

Chapter 24 Database Recovery

This chapter explains how to recover a database from failures.

Reference

Appendix A Operating System–Specific Information

This appendix lists all of 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 interested primarily in managing security should focus on the information presented in Part VII , "Database Security".

Conventions Used in This Manual

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

Special Icons

Special icons alert you to particular information within the body of this manual:

Suggestion: The lightbulb highlights suggestions and practical tips that could save time, make procedures easier, and so on.

Warning: The warning symbol highlights text that warns you of actions that could be particularly damaging or fatal to your operations.

OSDoc

Additional Information: The OSDoc icon refers you to the Oracle operating system–specific documentation for additional information.

Text of the Manual

The following sections describe the conventions used in the text of this manual.

UPPERCASE Characters

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

For example, "If you create a private rollback segment, the name must be included in the ROLLBACK_SEGMENTS parameter of the parameter file."

Italicized Characters

Italicized words within text are book titles or emphasized words.

Code Examples

SQL, Server Manager line mode, and SQL*Plus commands/statements appear separated from the text of paragraphs 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 words in example statements indicate the keywords within Oracle SQL. When you issue statements, however, keywords are not case sensitive.

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 comments as an Oracle customer. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of this manual is a Reader's Comment Form which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form has been used or you would like to contact us, please contact us at the following address:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

Contents

PART I WHAT IS ORACLE?

Chapter 1 **Introduction to the Oracle Server 1–1**

Databases and Information Management 1–2
The Oracle Server 1–4
Structured Query Language (SQL) 1–4
Database Structure 1–5
An Oracle Instance 1–5
Database Structure and Space Management 1–7
Relational Database Management Systems 1–7
Logical Database Structures 1–8
Physical Database Structures 1–15
The Data Dictionary 1–16
Oracle Server Architecture 1–17
Memory Structures and Processes 1–17
Memory Structures 1–18
Processes 1–20
The Program Interface 1–22
An Example of How Oracle Works 1–23
Data Access 1–24
SQL—The Structured Query Language 1–24
Transactions 1–25
PL/SQL 1–27
Data Integrity 1–28
Data Concurrency and Consistency 1–30

Concurrency 1–30
Read Consistency 1–31
Locking 1–32
Database Security 1–33
Security Mechanisms 1–34
Trusted Oracle 1–39
Database Backup and Recovery 1–40
Why Is Recovery Important? 1–40
Types of Failures 1–40
Structures Used for Recovery 1–42
Basic Recovery Steps 1–44
Distributed Processing and Distributed Databases 1–45
Client/Server Architecture: Distributed Processing 1–45
Distributed Databases 1–45
Table Replication 1–47
Oracle and SQL*Net 1–48

PART II **BASIC DATABASE OPERATION**

Chapter 2 **Database and Instance Startup and Shutdown 2–1**
Introduction to Database Startup and Database Shutdown 2–2
Connecting with Administrator Privileges 2–2
Database and Instance Startup 2–3
Starting an Instance 2–3
Mounting a Database 2–3
Opening a Database 2–4
Database and Instance Shutdown 2–5
Closing a Database 2–5
Dismounting a Database 2–6
Shutting Down an Instance 2–6
Parameter Files 2–6
An Example of a Parameter File 2–7
Changing Parameter Values 2–7

PART III DATABASE STRUCTURES

Chapter 3 Data Blocks, Extents, and Segments

Data Blocks, Extents, and Segments 3–1
The Relationships Between Data Blocks, Extents, and Segments 3–2
Data Blocks 3–2
Extents 3–3
Segments 3–3
Data Blocks 3–3
Data Block Format 3–3
An Introduction to PCTFREE, PCTUSED, and Row Chaining 3–5
Extents 3–10
When Extents Are Allocated for Segments 3–10
How Extents Are Allocated for Segments 3–13
When Extents Are Deallocated 3–14
Determining Sizes and Limits of Segment Extents 3–15
Segments 3–15
Data Segments 3–16
Index Segments 3–16
Rollback Segments 3–16
Temporary Segments 3–28
Operations Requiring Temporary Segments 3–28
How Temporary Segments Are Allocated 3–28

Chapter 4 Tablespaces and Datafiles

Tablespaces and Datafiles 4–1
An Introduction to Tablespaces and Datafiles 4–2
Tablespaces 4–3
The SYSTEM Tablespace 4–4
Allocating More Space for a Database 4–4
Online and Offline Tablespaces 4–6
Read–Only Tablespaces 4–8
Temporary Tablespaces 4–9
Datafiles 4–10
Datafile Contents 4–10
Size of Datafiles 4–11
Offline Datafiles 4–11

Chapter 5 **Schema Objects** 5–1
Overview of Schema Objects 5–2
Tables 5–3
How Table Data Is Stored 5–3
Nulls 5–6
Default Values for Columns 5–7
Views 5–8
Storage for Views 5–9
How Views Are Used 5–10
The Mechanics of Views 5–11
Dependencies and Views 5–12
Updatable Join Views 5–12
Partition Views 5–13
The Sequence Generator 5–16
Synonyms 5–17
Indexes 5–18
Unique and Non–Unique Indexes 5–19
Composite Indexes 5–19
Indexes and Keys 5–20
How Indexes Are Stored 5–20
Clusters 5–23
Performance Considerations 5–25
Format of Clustered Data Blocks 5–26
The Cluster Key 5–26
The Cluster Index 5–27
Hash Clusters 5–27
How Data Is Stored in a Hash Cluster 5–28
Hash Key Values 5–30
Hash Functions 5–31
Allocation of Space for a Hash Cluster 5–32

Chapter 6 **Datatypes** 6–1
Oracle Datatypes 6–2
Character Datatypes 6–2
NUMBER Datatype 6–4
DATE Datatype 6–6
LONG Datatype 6–7
RAW and LONG RAW Datatypes 6–8
ROWIDs and the ROWID Datatype 6–9
The MLSLABEL Datatype 6–11
Summary of Oracle Datatype Information 6–12

ANSI, DB2, and SQL/DS Datatypes 6–14
Data Conversion 6–15
Rule 1: Assignments 6–15
Rule 2: Expression Evaluation 6–16

Chapter 7 **Data Integrity 7–1**
Definition of Data Integrity 7–2
Types of Data Integrity 7–2
How Oracle Enforces Data Integrity 7–3
An Introduction to Integrity Constraints 7–5
Advantages of Integrity Constraints 7–5
The Performance Cost of Integrity Constraints 7–6
Types of Integrity Constraints 7–7
NOT NULL Integrity Constraints 7–7
UNIQUE Key Integrity Constraints 7–8
PRIMARY KEY Integrity Constraints 7–10
Referential Integrity and FOREIGN KEY (Referential) Integrity Constraints 7–11
Actions Defined by Referential Integrity Constraints 7–15
CHECK Integrity Constraints 7–16
The Mechanisms of Constraint Checking 7–17
Default Column Values and Integrity Constraint Checking . 7–19

Chapter 8 **The Data Dictionary 8–1**
An Introduction to the Data Dictionary 8–2
The Structure of the Data Dictionary 8–3
SYS, the Owner of the Data Dictionary 8–3
How the Data Dictionary Is Used 8–3
How Oracle and Other Oracle Products Use the Data Dictionary 8–4
How Oracle Users Can Use the Data Dictionary 8–5
The Dynamic Performance Tables 8–7

PART IV SYSTEM ARCHITECTURE

Chapter 9 **Memory Structures and Processes 9–1**

An Oracle Instance 9–2

Process Structure 9–3

Single–Process Oracle Instance 9–3

Multiple–Process Oracle Instance 9–4

Oracle Memory Structures 9–15

Virtual Memory 9–15

Software Code Areas 9–16

System Global Area (SGA) 9–16

Program Global Area (PGA) 9–26

Sort Areas 9–28

Sort Direct Writes 9–29

Variations in Oracle Configuration 9–29

Connections, Sessions, and User Processes 9–30

Oracle Using Combined User/Server Processes 9–31

Oracle Using Dedicated Server Processes 9–32

The Multi–Threaded Server 9–34

Examples of How Oracle Works 9–39

An Example of Oracle Using Dedicated Server Processes . . 9–39

An Example of Oracle Using the Multi–Threaded Server . . 9–40

The Program Interface 9–41

Program Interface Structure 9–41

The Program Interface Drivers 9–42

Operating System Communications Software 9–42

Chapter 10 **Data Concurrency 10–1**

Data Concurrency in a Multi–user Environment 10–2

General Concurrency Issues 10–2

Locking Mechanisms 10–3

How Oracle Controls Data Concurrency 10–5

Multiversion Concurrency Control 10–5

Statement Level Read Consistency 10–7

Transaction Level Read Consistency 10–7

Oracle Isolation Levels 10–8

Setting the Isolation Level 10–8

Additional Considerations for Serializable Isolation 10–10

Comparing Read Committed and Serializable Isolation 10–12

How Oracle Locks Data 10–16

Transactions and Data Concurrency 10–17
Types of Locks 10–18
Data Locks 10–19
DDL Locks (Dictionary Locks) 10–26
Internal Locks and Latches 10–27
Explicit (Manual) Data Locking 10–29
Oracle Lock Management Services 10–36

PART V **DATA ACCESS**

Chapter 11 **SQL and PL/SQL 11–1**
Structured Query Language (SQL) 11–2
SQL Statements 11–2
Identifying Non–Standard SQL 11–5
Recursive SQL 11–6
Cursors 11–6
Shared SQL 11–6
What Is Parsing? 11–6
PL/SQL 11–7
How PL/SQL Executes 11–8
Language Constructs for PL/SQL 11–9

Chapter 12 **Transaction Management 12–1**
Introduction to Transactions 12–2
Statement Execution and Transaction Control 12–3
Statement–Level Rollback 12–4
Oracle and Transaction Management 12–4
Committing Transactions 12–5
Rolling Back Transactions 12–6
Savepoints 12–6
Discrete Transaction Management 12–7

Chapter 13 **The Optimizer 13–1**
What Is Optimization? 13–2
Execution Plans 13–2
Oracle's Approaches to Optimization 13–6
Histograms 13–8
How Oracle Optimizes SQL Statements 13–9
Types of SQL Statements 13–10

Choosing Access Paths 13–11
Optimizing Distributed Statements 13–17

PART VI PROGRAMMATIC CONSTRUCTS

Chapter 14 **Procedures and Packages 14–1**
An Introduction to Stored Procedures and Packages 14–2
Stored Procedures and Functions 14–2
Packages 14–3
PL/SQL 14–5
Procedures and Functions 14–5
How Procedures Are Used 14–6
Applications for Procedures 14–7
Anonymous PL/SQL Blocks vs. Stored Procedures 14–8
Standalone Procedures vs. Package Procedures 14–8
Dependency Tracking for Stored Procedures 14–8
Packages 14–9
Applications for Packages 14–12
Dependency Tracking for Packages 14–14
How Oracle Stores Procedures and Packages 14–14
Compiling Procedures and Packages 14–14
Storing the Compiled Code in Memory 14–14
Storing Procedures or Packages in Database 14–14
How Oracle Executes Procedures and Packages 14–15
Verifying User Access 14–15
Verifying Procedure Validity 14–16
Executing a Procedure 14–16

Chapter 15 **Database Triggers 15–1**
An Introduction to Triggers 15–2
How Triggers Are Used 15–3
A Cautionary Note about Trigger Use 15–3
Database Triggers vs. Oracle Forms Triggers 15–4
Triggers vs. Declarative Integrity Constraints 15–5
Parts of a Trigger 15–5
Triggering Event or Statement 15–6
Trigger Restriction 15–7
Trigger Action 15–7
Types of Triggers 15–7
Trigger Execution 15–11

The Execution Model for Triggers and Integrity Constraint Checking 15–11
Data Access for Triggers 15–13
Storage for Triggers 15–14
Execution of Triggers 15–15
Dependency Maintenance for Triggers 15–15

Chapter 16 **Dependencies Among Schema Objects 16–1**
An Introduction to Dependency Issues 16–2
Compiling Views and PL/SQL Program Units 16–4
Advanced Dependency Management Topics 16–6
Dependency Management and Non–Existent Schema Objects 16–6
Shared SQL Dependency Management 16–7
Package Invalidations and Session State 16–8
Local and Remote Dependency Management 16–8

PART VII **DATABASE SECURITY**

Chapter 17 **Database Access 17–1**
Schemas, Database Users, and Security Domains 17–2
User Authentication 17–3
Authenticating Users Using the Operating System 17–3
Authenticating Users Using Network Authentication 17–4
Authenticating Users Using the Oracle Database 17–4
Password Encryption while Connecting 17–4
Database Administrator Authentication 17–4
User Tablespace Settings and Quotas 17–6
Default Tablespace 17–6
Temporary Tablespace 17–6
Tablespace Access and Quotas 17–6
The User Group PUBLIC 17–7
User Resource Limits and Profiles 17–8
Types of System Resources and Limits 17–9
Profiles 17–11
Licensing 17–12
Concurrent Usage Licensing 17–12
Named User Licensing 17–14

Chapter 18 **Privileges and Roles 18–1**
Privileges 18–2
System Privileges 18–2
Object Privileges 18–3
Roles 18–10
Common Uses for Roles 18–10
The Mechanisms of Roles 18–11
Granting and Revoking Roles 18–12
Who Can Grant or Revoke Roles? 18–12
Who Can Grant or Revoke Roles? 18–12
Who Can Grant or Revoke Roles? 18–12
Naming Roles 18–12
Security Domains of a Role and a User Granted Roles 18–12
Data Definition Language Statements and Roles 18–13
Predefined Roles 18–14
The Operating System and Roles 18–14
Roles in a Distributed Environment 18–14

Chapter 19 **Auditing 19–1**
Introduction to Auditing 19–2
Auditing Features 19–2
Auditing Mechanisms 19–4
Statement Auditing 19–6
Privilege Auditing 19–7
Object Auditing 19–7
Object Audit Options for Views and Procedures 19–8
Focusing Statement, Privilege, and Object Auditing 19–9
Auditing Successful and Unsuccessful Statement Executions 19–9
Auditing BY SESSION versus BY ACCESS 19–9
Auditing By User 19–11

PART VIII DISTRIBUTED PROCESSING AND DISTRIBUTED DATABASES

Chapter 20 **Client/Server Architecture** **20–1**
The Oracle Client/Server Architecture 20–2
Distributed Processing 20–2
SQL*Net 20–5
How SQL*Net Works 20–5

Chapter 21 **Distributed Databases** **21–1**
An Introduction to Distributed Databases 21–2
Clients, Servers, and Nodes 21–2
Site Autonomy 21–3
Schema Objects and Naming in a Distributed Database 21–4
Database Links 21–5
Statements and Transactions in a Distributed Database 21–6
Two–Phase Commit Mechanism 21–6
Transparency in a Distributed Database System 21–7
SQL*Net and Network Independence 21–8
Heterogeneous Distributed Database Systems 21–8
Replicating Data 21–10

PART IX DATABASE BACKUP AND RECOVERY

Chapter 22 **Recovery Structures** **22–1**
An Introduction to Database Recovery and Recovery Structures 22–2
Errors and Failures 22–2
Structures Used for Database Recovery 22–5
The Online Redo Log 22–6
Online Redo Log File Contents 22–6
How Online Redo Log Files Are Written 22–7
The Archived Redo Log 22–16
The Mechanics of Archiving 22–16
Archived Redo Log File Contents 22–17
Database Archiving Modes 22–18
Control Files 22–21
Control File Contents 22–21
Multiplexed Control Files 22–22
Survivability 22–23

Planning for Disaster Recovery 22–23
Standby Database 22–23

Chapter 23 **Database Backup 23–1**
An Introduction to Database Backups 23–2
Full Backups 23–2
Partial Backups 23–3
The Export and Import Utilities 23–8
Read–Only Tablespaces and Backup 23–8

Chapter 24 **Database Recovery 24–1**
Recovery Procedures 24–2
Recovery Features 24–2
An Introduction to Database Recovery 24–3
Database Buffers and DBWR 24–3
The Redo Log and Rolling Forward 24–3
Rollback Segments and Rolling Back 24–4
Performing Recovery in Parallel 24–5
What Situations Benefit from Parallel Recovery 24–5
Recovery Processes 24–5
Recovery from Instance Failure 24–7
Read–Only Tablespaces and Instance Recovery 24–7
Recovery from Media Failure 24–8
Read–Only Tablespaces and Media Recovery 24–8
Complete Media Recovery 24–10
Incomplete Media Recovery 24–14

Appendix A **Operating System–Specific Information A–1**

Index

PART

I

What Is Oracle?

CHAPTER

1

Introduction to the Oracle Server

I am Sir Oracle,
And when I ope my lips, let no dog bark!

Shakespeare: *The Merchant of Venice*

This chapter is an overview of the Oracle Server. It presents information that will help you use the rest of this book. It includes:

- Databases and Information Management
- The Oracle Server
- Database Structure and Space Management
- Oracle Server Architecture
- Data Access
- Data Concurrency and Consistency
- Database Security
- Database Backup and Recovery
- Distributed Processing and Distributed Databases

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 multi–user 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 (distributed processing) environments	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, potentially terabytes in size. To make efficient use of expensive hardware devices, it 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.
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.

Release 7.3 of the Oracle Server has been certified by the U.S. National Institute of Standards and Technology as 100% compliant with Entry Level of the ANSI/ISO SQL92 (Structured Query Language) standard. Oracle fully satisfies the requirements of the U.S. Government's FIPS127–2 standard and includes a "flagger" to highlight non–standard SQL usage.

Also, Oracle7 has been evaluated by the U.S. Government's National Computer Security Center (NCSC) as compliant with the Orange Book security criteria; the Oracle7 Server and Trusted Oracle7 comply with the C2 and B1 Orange Book levels, respectively, as well as with comparable European ITSEC security criteria.

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. As a result, the costs of coding and managing checks in many database applications are eliminated.

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.

portability
: Oracle software is ported to work 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.

connectibility
: Oracle software allows different types of computers and operating systems to share information across networks.

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 major section describes a different part of the overall architecture.

The Oracle Server

The Oracle Server is a 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 instance. The following sections describe the relationship between the database and the instance.

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. And 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 commands. You use data definition language (DDL) commands to set up the data. DDL commands include commands to creating and altering databases and tables.

You can update, delete, or retrieve data in a table with data manipulation commands (DML). DML commands include commands to alter and fetch dat. The most common SQL command is the SELECT command, which allows you to retrieve data from the database.

In addition to SQL commands, the Oracle Server has a procedural language called PL/SQL. PL/SQL enables the programmer to program SQL statements. It allows you to control the flow of a SQL program, to use variables, and to write error–handling procedures.

Database Structure

An Oracle database has both a physical and a logical structure. Because the physical and logical server structure are separate, the physical storage of data can be managed without affecting the access to logical storage structures.

Physical Database Structure An Oracle database's physical structure is determined by the operating system files that constitute the database. Each Oracle database is made of three types of files: one or more datafiles, two or more redo log files, and one or more control files. The files of an Oracle database provide the actual physical storage for database information.

Logical Database Structure An Oracle database's logical structure is determined by

- one or more tablespaces. (A tablespace is a logical area of storage explained later in this chapter.)
- the database's schema objects. A *schema* is a collection of 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.

The logical storage structures, including tablespaces, segments, and extents, dictate how the physical space of a database is used. The schema objects and the relationships among them form the relational design of a database.

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 a 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.* Figure 1 – 1 illustrates a multiple process Oracle instance.

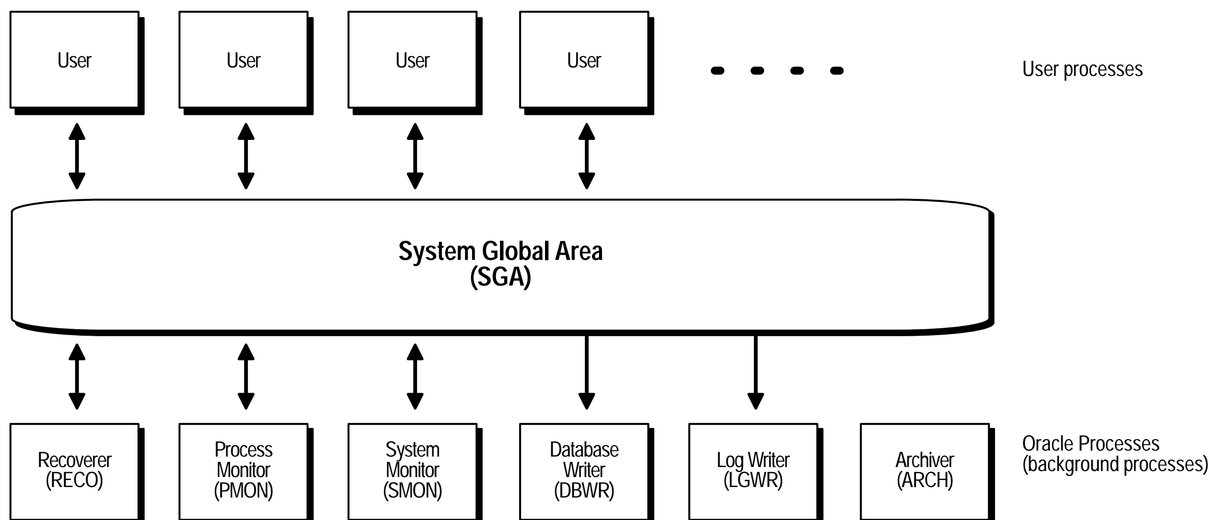


Figure 1 – 1 An Oracle Instance

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 Server Manager).

Oracle processes are server processes that perform work for user processes and background processes that perform maintenance work for the Oracle Server.

Communications Software and SQL*Net

If the user and server processes are on different computers of a network or if the user processes connect to shared server processes through dispatcher processes, the user process and server process communicate using SQL*Net. *Dispatchers* are optional background processes, present only when a multi–threaded server configuration is used. *SQL*Net* is Oracle's interface to standard communications protocols that allows for the proper transmission of data between computers. See "Oracle and SQL*Net" on page 1–48.

The Oracle Parallel Server: Multiple Instance Systems

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.

Database Structure and Space Management

This section describes the architecture of an Oracle database, including the physical and logical structures that make up a database. This discussion provides an understanding of Oracle's solutions to controlled data availability, the separation of logical and physical data structures, and fine–grained control of disk space management.

Relational Database Management Systems

Database management systems have evolved from hierarchical to network to relational models. Today, the most widely accepted database model is 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

An Oracle *database* is a collection of data that is treated as a unit. The general purpose of a database is to store and retrieve related information. The database has *logical structures* and *physical structures.*

Open and Closed Databases

An Oracle database can be *open* (accessible) or *closed* (not accessible). In normal situations, the database is open and available for use. However, the database is sometimes closed for specific administrative functions that require the database's data to be unavailable to users.

Logical Database Structures

The following sections explain logical database structures, including tablespaces, schema objects, data blocks, extents, and segments.

Tablespaces

A database is divided into logical storage units called *tablespaces.* A tablespace is used to group related logical structures together. For example, tablespaces commonly group all of an application's objects to simplify certain 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 – 2.

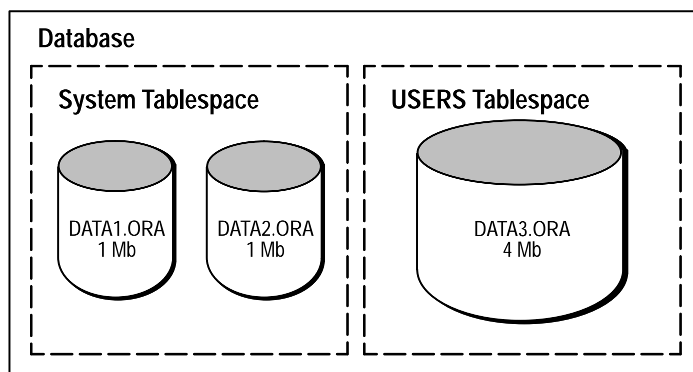


Figure 1 – 2 Databases, Tablespaces, and Datafiles

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 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. (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. For more information, see "Data Integrity" on page 1–28.

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 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 commands 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. For information about updatable join views, see "Modifying a Join View" in the *Oracle7 Server Application Developer's Guide.*

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 The term "program unit" is used in this manual to refer to stored procedures, functions, and packages.

> **Note:** The information in this section applies only to Oracle with the procedural option installed (PL/SQL.).

A *procedure* or *function* is a set of SQL and PL/SQL (Oracle's procedural language extension to SQL) statements grouped together as an executable unit to perform a specific task. For more information about SQL and PL/SQL, see "Data Access" on page 1–24.

Procedures and functions allow you to combine the ease and flexibility of SQL with the procedural functionality of a structured programming language. Using PL/SQL, such procedures and functions can be defined and stored in the database for continued use. Procedures and functions are identical, except that functions always return a single value to the caller, while procedures do not return a value to the caller.

Packages provide a method of encapsulating and storing related procedures, functions, and other package constructs together as a unit in the database. While packages provide the database administrator or application developer organizational benefits, they also offer increased functionality and database performance.

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

- mask the real name and owner of an object
- provide public access to an 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, Clusters, and Hash Clusters Indexes, clusters, and hash clusters are optional structures associated with tables, which can be created to increase the performance of data retrieval.

Indexes are 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.

Clusters are an optional method of storing table data. Clusters are groups of one or more tables physically stored together because they share common columns and are often used together. Because related rows are physically stored together, disk access time improves.

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). Figure 1 – 3 illustrates how clustered and non–clustered data is physically stored.

Clusters also can improve performance of data retrieval, depending on data distribution and what SQL operations are most often performed on the clustered 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.

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 hash 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 specified rows.

Database Links A *database link* is a named 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. Also see "Distributed Databases" on page 1–45.

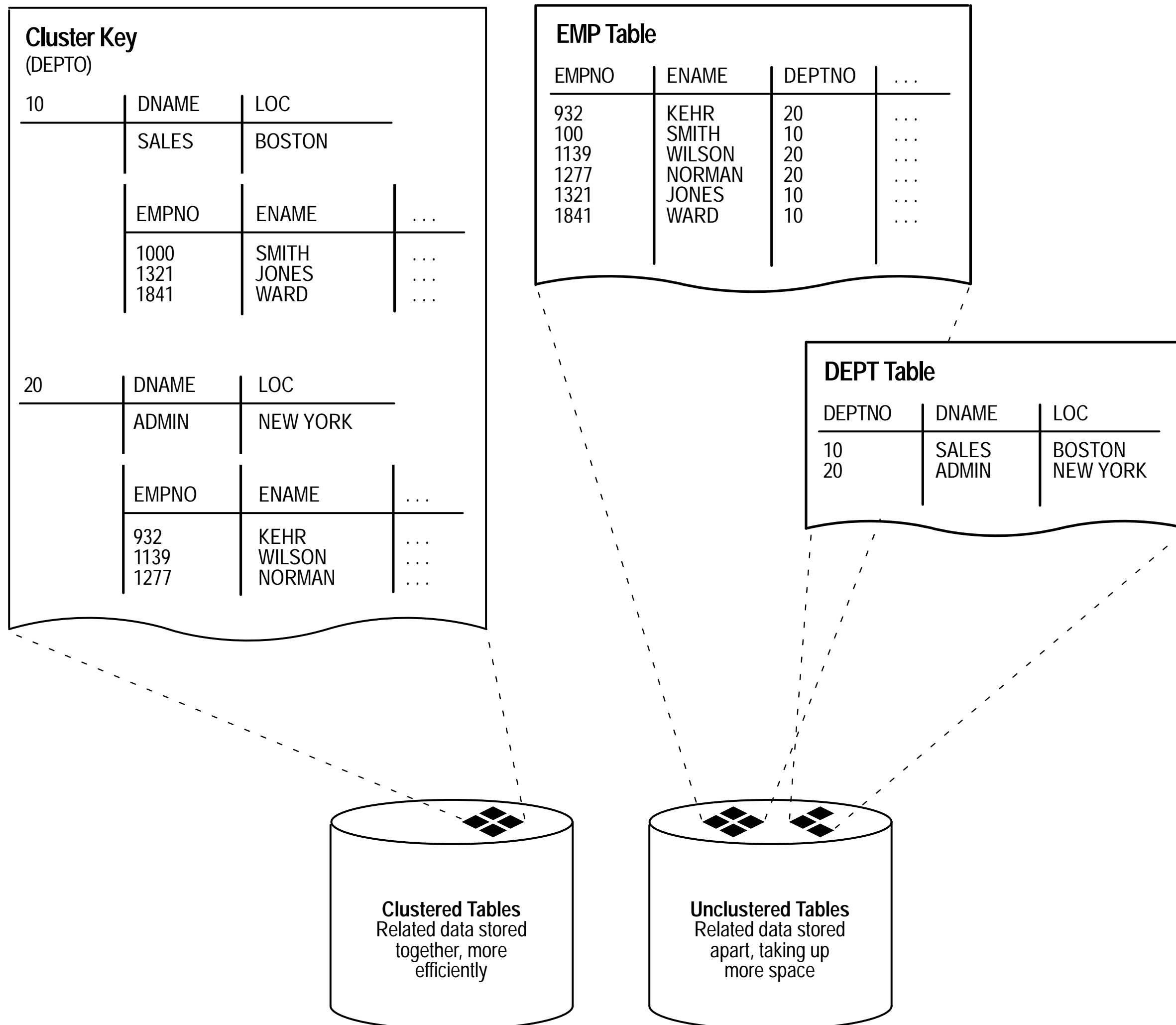


Figure 1 – 3 Clustered and Unclustered Tables

Data Blocks, Extents, and Segments

Oracle allows fine–grained control of disk space usage through the logical storage structures, including data blocks, extents, and segments. For more information on these, see Chapter 3 of this manual.

Oracle Data Blocks At the finest level of granularity, an Oracle database's 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 the following:

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. 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.
Rollback Segment	One or more rollback segments are created by the database administrator for a database to temporarily store "undo" information. This information is used: • to generate read–consistent database information. See "Read–Consistency" on page 1–31. • during database recovery. See "Database Backup and Recovery" on page 1–40. • 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.

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 following are characteristics of datafiles:

- A datafile can be associated with only one database.
- Database files 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 DBWR background process of Oracle. (For more information about Oracle's memory and process structures and the algorithm for writing database data to the datafiles, see "Oracle Server Architecture" on page 1–17.)

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.* 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 abruptly 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 "Database Backup and Recovery" on page 1–40.

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 a database's 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 database's control file is automatically modified by Oracle to reflect the change.

A database's control file is also used if database recovery is necessary. See "Database Backup and Recovery" on page 1–40.

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 being used

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.

Oracle Server Architecture

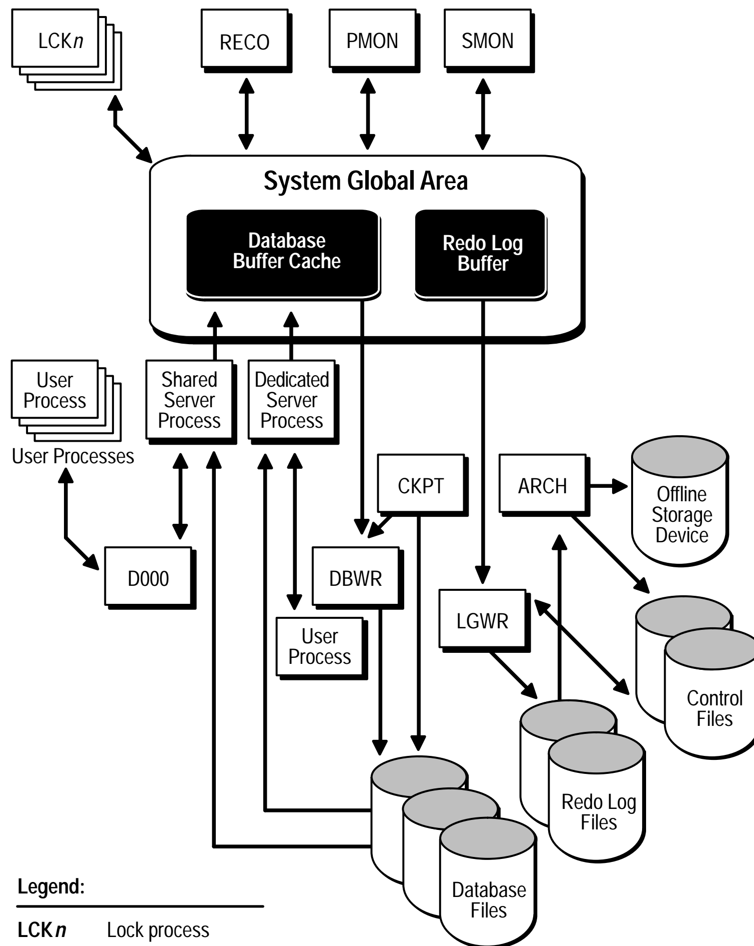
The following section discusses the memory and process structures used by an Oracle Server to manage a database. Among other things, the architectural features discussed in this section provide an understanding of Oracle's capabilities to support

- many users concurrently accessing a single database
- the high performance required by concurrent multi–user, multi–application database systems

Memory Structures and Processes

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 – 4 shows a typical variation of the Oracle Server memory and process structures.



Legend:

LCK*n*	Lock process
RECO	Recoverer process
PMON	Process monitor
SMON	System monitor
CKPT	Checkpoint
ARCH	Archiver
DBWR	Database writer
LGWR	Log writer

Figure 1 – 4 Memory Structures and Processes of Oracle

Memory Structures

Oracle creates and uses memory structures to complete several jobs. For example, memory is used to store program code being executed and data that is shared among users. Several 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 sections explain each in detail.

System Global Area (SGA)

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. (See "Background Processes" on page 1–21 for information about the Oracle background processes and "An Oracle Instance" on page 1–5 for information about Oracle instances.)

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.

Database Buffer Cache *Database buffers* of the system global area store the most recently used blocks of database data; the set of database buffers in an instance is the *database buffer cache.* These buffers can contain modified data that has not yet been permanently written to disk. Because the most recently (and often the most frequently) used data is kept in memory, less disk I/O is necessary and performance is increased.

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 (see "SQL Statements" on page 1–24). 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.

Cursors A *cursor* is a handle (a name or pointer) for the memory associated with a specific statement. 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.

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.

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 Server Manager). The user process also manages the communication with the server processes. User processes communicate with the server processes through the program interface, described later in this section.

Oracle Processes

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.

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 Oracle background processes constitute an Oracle instance. (See "System Global Area (SGA)" on page 1–19 for information about the system global area and "An Oracle Instance" on page 1–5 for information about Oracle instances.)

Each Oracle instance may use several background processes. The names of these processes are DBWR, LGWR, CKPT, SMON, PMON, ARCH, RECO, D*nnn* and LCK*n*. Each background process is described in the following sections.

Database Writer (DBWR) The *Database Writer* writes modified blocks from the database buffer cache to the datafiles. Because of the way Oracle performs logging, DBWR does not need to write blocks when a transaction commits (see "Transactions" on page 1–25). Instead, DBWR is optimized to minimize disk writes. In general, DBWR 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.

Log Writer (LGWR) The *Log Writer* writes redo log entries to disk. Redo log data is generated in the redo log buffer of the system global area. As transactions commit and the log buffer fills, LGWR writes redo log entries into an online redo log file.

Checkpoint (CKPT) At specific times, all modified database buffers in the system global area are written to the datafiles by DBWR; this event is called a checkpoint. The *Checkpoint* process is responsible for signalling DBWR at checkpoints and updating all the datafiles and control files of the database to indicate the most recent checkpoint. CKPT is optional; if CKPT is not present, LGWR assumes the responsibilities of CKPT.

System Monitor (SMON) The *system monitor* performs instance recovery at instance startup. In a multiple instance system (one that uses the Parallel Server), SMON of one instance can also 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 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 (ARCH) The *archiver* copies the online redo log files to archival storage when they are full. ARCH is active only when a database's redo log is used in ARCHIVELOG mode. (See "The Redo Log" on page 1–42).

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 (LCK*n*) Up to ten *lock* processes (LCK0, . . ., LCK9) are used for inter–instance locking when the Oracle Parallel Server is used; see "The Oracle Parallel Server: Multiple Instance Systems" on page 1–6 for more information about this configuration.

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 used to run 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 SQL*Net driver.
3. The server is running the proper SQL*Net 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 DBWR 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.

Data Access

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

- 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 Statements (DDL) *DDL statements* define, maintain, and drop 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 “Database Security” on page 1–33.)

Data Manipulation Statements (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. (See “Transactions” on page 1–25) Examples include COMMIT, ROLLBACK, and SAVEPOINT.

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 commands change the properties of the Oracle Server instance. The only system control command 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.

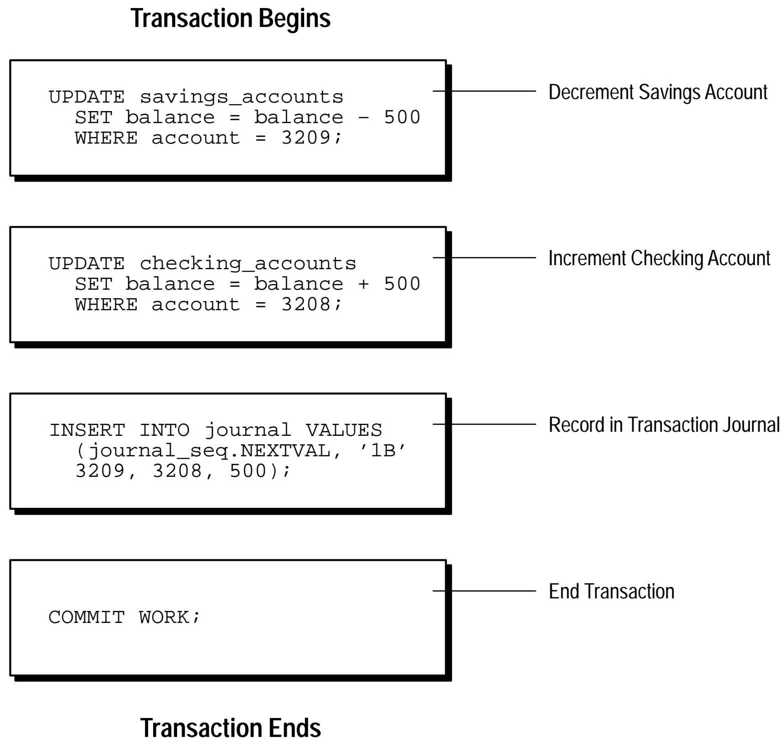


Figure 1 – 5 A Banking Transaction

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 comprise 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, 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).

Database Triggers

Oracle allows you to write procedures that are automatically executed as a result of an insert in, update to, or delete from a table. 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.

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 The term "key" is used in the definitions of several types of integrity constraints. A *key* is the column or set of columns included in the definition of certain types of integrity constraints. Keys describe the relationships between the 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

Centralized actions can be defined using a non–declarative approach (writing PL/SQL code) with database triggers. A *database trigger* is a stored procedure that is fired (implicitly executed) when an INSERT, UPDATE, or DELETE statement is issued against the associated table. Database triggers can be used to customize a database management system with such features as value–based auditing and the enforcement of complex security checks and integrity rules. For example, a database trigger might be created to allow a table to be modified only during normal business hours.

> **Note:** While database triggers allow you to define and enforce integrity rules, 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.

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 multi–user 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 non–existent or negligible to each user. All DML 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. As discussed in "Data Consistency Using Transactions" on page 1–27, it is the application designer's responsibility to ensure that transactions fully exploit these 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, which 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

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.

Database Security

Multi–user database systems, such as Oracle, include security features that control how a database is accessed and used. For example, security mechanisms do the following:

- prevent unauthorized database access
- prevent unauthorized access to schema objects
- control disk usage
- control system resource usage (such as CPU time)
- audit user actions

Associated with each database user is a *schema* by the same name. A schema is a logical collection of objects (tables, views, sequences, synonyms, indexes, clusters, procedures, functions, packages, and database links). 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 the objects of a user
- 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
- which system operations a user can perform

Data security includes the mechanisms that control the access and use of the database at the 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 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 an 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 – 6 illustrates the relationships of the different Oracle security facilities, and the following sections provide an overview of users, privileges, and roles.

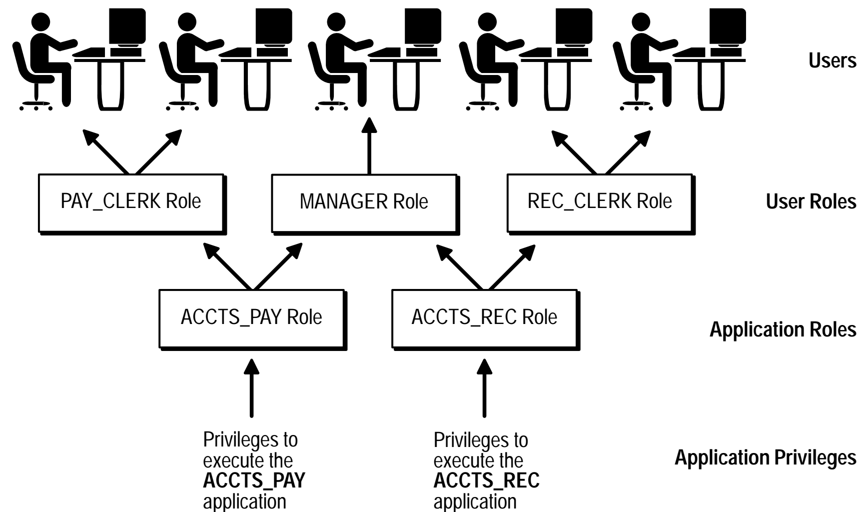


Figure 1 – 6 Oracle Security Features

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
- tablespace quotas (available disk space) for the user
- 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)
- right to create a table in your schema
- right to select rows from someone else's table
- right to execute someone else's stored procedure

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

System Privileges *System privileges* allow users to perform a particular systemwide action or a particular action on a particular type of 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.

Object Privileges *Object privileges* allow users to perform a particular action on a specific 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 granted to a role. And then the database administrator can 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 object, the user's default tablespace is used if the user has the privilege to create the object and a quota in the specified default tablespace. The default tablespace feature provides Oracle with information to direct space usage in situations where object 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
  - available to a single call to Oracle made by a SQL statement
- amount of logical I/O
  - available to the user's session
  - available to 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

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 object auditing.

statement auditing

Statement auditing is the auditing of specific SQL statements without regard to specifically named 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 objects. Privilege auditing can be broad and audit all users or can be focused to audit only selected users.

object auditing — *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.

Trusted Oracle

Trusted Oracle is Oracle Corporation's multilevel secure database management system product. It is designed to provide the high level of secure data management capabilities required by organizations processing sensitive or classified information. Trusted Oracle is compatible with Oracle base products and applications, and it supports all of the functionality of standard Oracle.

In addition, Trusted Oracle enforces *mandatory access control* (also called *MAC)* across a wide range of multilevel secure operating system environments. Mandatory access control is a means of restricting access to information based on *labels*. A user's label indicates what information a user is permitted to access and the type of access (read or write) that the user is allowed to perform. An object's label indicates the sensitivity of the information that the object contains. A user's label must meet certain criteria, determined by MAC policy, in order for him/her to be allowed to access a labeled object. Because this type of access control is always enforced above any discretionary controls implemented by users, this type of security is called "mandatory".

See the *Trusted Oracle7 Server Administrator's Guide* for more information.

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 for 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 (system global area and background processes) 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 *instance recovery.* Instance recovery is automatically performed by Oracle when the instance is restarted. 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 necessary database backups.

The Redo Log

As described in "Redo Log Files" on page 1–15, 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 be comprised 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 committed changes made to the database. Whenever a transaction is committed, the corresponding redo entries temporarily stored in redo log buffers of the system global area are written to an online redo log file by the background process LGWR.

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 is operated in NOARCHIVELOG mode, the database can be completely recovered from instance failure, but not from a disk failure. Additionally, 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.

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 This feature is similar to the multiplexed redo log feature: a number of identical control files may be maintained by Oracle, which updates all of them simultaneously.

Rollback Segments

As described in "Segments" on page 1–14, 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.

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.

Full Backups A full backup is an operating system backup of all datafiles, online redo log files, and the control file that constitutes an

Oracle database. Full backups are 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 DBWR writes database buffers to datafiles, at any given point in time, a datafile may contain some data blocks tentatively modified by uncommitted transactions and may not contain some blocks modified by committed transactions. Therefore, two potential situations can result after a failure:

- 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 needed redo information is online, Oracle performs this recovery step 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.

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.

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.

Server The server portion runs Oracle software and handles the functions required for concurrent, shared data access. The server portion receives and processes SQL and PL/SQL statements originating 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.

Distributed Databases

> **Note:** The information in this section regarding distributed updates and two–phase commit applies only for those systems using Oracle with the distributed option.

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 non–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 non–distributed environment. Oracle provides this assurance using the transaction model and a *two–phase commit mechanism*. As in non–distributed 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

> **Note:** The information in this section applies only to Oracle with the distributed or advanced replication options.

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 snapshots or replicated master tables. Replicated master tables require the replication option. For more

information about replicating data, see *Oracle7 Server Distributed Systems, Volume II.*

Oracle and SQL*Net

*SQL*Net* 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 SQL*Net. SQL*Net 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 SQL*Net, 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.

PART

II

Basic Database Operation

CHAPTER

2

Database and Instance Startup and Shutdown

Greetings, Prophet;
The Great Work begins:
The Messenger has arrived.

Tony Kushner: *Angels in America*, Part I

This chapter explains the concepts involved in starting and stopping an Oracle instance and database. It includes:

- Introduction to Database Startup and Database Shutdown
- Database and Instance Startup
- Database and Instance Shutdown
- Parameter Files

If you are using Trusted Oracle, refer to the *Trusted Oracle7 Server Administrator's Guide* for more information about starting up and shutting down in that environment.

Introduction to Database Startup and Database Shutdown

An Oracle database may not always be available to all users. The database administrator can start up a database so that it is open. When a database is *open*, users can access the information that it contains. 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.

Only a database administrator can open or close a database. Normal users do not have control over the current status of an Oracle database. Security for database startup and shutdown is controlled via connections to Oracle with administrator privileges.

Connecting with Administrator Privileges

Database startup and shutdown are powerful administrative options and are protected by the ability to connect to Oracle with administrator privileges.

Depending on the operating system, one of the following prerequisites is required to connect to Oracle with administrator privileges:

- The user's operating system account has operating system privileges that allow him/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.

In addition, users can connect with administrator privileges only to dedicated servers (not shared servers).

When you connect with administrator privileges, you are placed in the schema owned by SYS.

These requirements provide extra security to prevent unauthorized users from starting up or shutting down any Oracle databases.

For more information about password files and authentication schemes for database administrators, see Chapter 17, "Database Access".

OSDoc

Additional Information: For information on how administrator privileges work on your operating system, see your operating system–specific Oracle documentation.

Database and Instance Startup

There are three steps to starting a database and making it available for systemwide use:

1. Start an instance.
2. Mount the database.
3. Open the database.

Starting an Instance

Starting an instance includes the allocation of an SGA — a shared area of memory used for database information — and creation of the background processes. Instance startup anticipates a database being mounted by the instance. If an instance has been started but not yet mounted, no database is associated with these memory structures and processes.

Before an instance actually is created, Oracle reads a parameter file, which determines the instance initialization. This file includes parameters that control such things as the size of the SGA, the name of the database to which the instance can connect, and so on.

> **Note:** See "Parameter Files" on page 2–6 for more information about parameter files. See Chapter 9, "Memory Structures and Processes", for more information about the terms "SGA", "background processes", and "instance".

Restricted Mode of Instance Startup

You can start an instance in or alter an existing instance to be in restricted mode. This limits 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, an instance might not be shut down "cleanly", for example, one of the instance's processes might not be killed. In such situations, the database might return an error during normal instance startup. To resolve this problem, the database administrator must kill all remnant Oracle processes of the previous instance and then start the new instance.

Mounting a Database

Mounting a database associates a database with a previously started instance. After an instance mounts a database, the database remains closed and is accessible only to database administrators. The database administrator might want to start an instance and only mount the database to complete specific maintenance operations.

When an instance mounts a database, the instance finds the control files and opens them. The control files are specified in the CONTROL_FILES initialization parameter in the parameter file used to start the instance. Once the database's control files are opened, Oracle reads them to get the names of the database's datafiles and redo log files.

Modes of Mounting a Database with the Parallel Server

If Oracle allows multiple instances to mount the same database concurrently, the DBA can choose whether to run the database in exclusive or parallel mode.

Exclusive Mode If the first instance that mounts a database does so in exclusive mode, only that instance can mount the database. Versions of Oracle that do not support the Parallel Server option only allow an instance to mount a database in exclusive mode.

Parallel Mode If the first instance that mounts a database is started in parallel mode (also called "shared mode"), other instances that are started in parallel mode can also mount the database. The number of instances that can mount the database is subject to a predetermined maximum. See *Oracle7 Parallel Server Concepts & Administration* for more information about the use of multiple instances with a single database.

Opening a Database

Opening a mounted database makes the database available for normal database operations. Any valid user can connect to the database and access its information once it is open. Usually, the 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. See "Online and Offline Tablespaces" on page 4–6.

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 database files before you can open the database.

Instance Recovery

If the database was shut down either because the database administrator aborted its instance or because a power failure occurred while the database was running, Oracle automatically performs instance recovery when the database is reopened. See Chapter 24, "Database Recovery", for complete information concerning instance recovery.

Rollback Segment Acquisition

As an instance opens a database, the instance attempts to acquire one or more rollback segments. See "Instances and Types of Rollback Segments" on page 3–23.

Resolution of In–Doubt Distributed Transaction

Assume that a database is abruptly shut down (for example, a power failure occurs or the instance is aborted) and one or more distributed transactions have not been committed or rolled back. When you reopen the database and instance recovery is complete, the RECO background process automatically, immediately, and consistently resolves any distributed transactions that have been committed or rolled back. For information about distributed transactions, see Chapter 21, "Distributed Databases". For information about recovery from failures associated with distributed transactions, see *Oracle7 Server Distributed Systems, Volume I.*

Database and Instance Shutdown

There are three steps to shutting down an instance and the database to which it is connected:

1. Close the database.
2. Dismount the database.
3. Shut down the instance.

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

Closing a Database

The first step of database shutdown is closing the 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. After this operation, Oracle closes all online datafiles and online redo log files. Any offline datafiles of any offline tablespaces will have been closed already. When you subsequently reopen the database, the tablespace that was offline and its datafiles remain offline and closed, respectively. 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 instance 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 instance recovery is performed when the database is reopened.

Dismounting a Database

The second step accomplished during database shutdown is dismounting or disassociating the database from an instance. After you dismount a database, only an 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 killed. When remnants of previous instances exist, subsequent instance startup most likely will fail. To handle this problem, 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.

Parameter Files

To start an instance, Oracle must read a parameter file.A *parameter file* is a text file containing a list of instance configuration parameters. You set these parameters to particular values and to initialize many of the memory and process settings of an Oracle instance. Among other things, the parameters of this file tell Oracle the following:

- 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 = ORA7PROD
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
```

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

Most parameters belong to one of the following groups:

- parameters that name things (such as files)
- parameters that set limits (such as maximums)
- parameters affecting capacity, called *variable parameters* (such as the DB_BLOCK_BUFFERS parameter, which specifies the number of data blocks to allocate in the computer's memory for the SGA)

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.

PART

III

Database Structures

CHAPTER

3

Data Blocks, Extents, and Segments

He was not merely a chip of the old block, but the old block itself.

Edmund Burke: *On Pitt's first speech*

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

- The Relationship Between Data Blocks, Extents, and Segments
- Data Blocks
- Extents
- Segments
- Temporary Segments

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for more information about storage in that environment.

The Relationships Between Data Blocks, Extents, and Segments

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

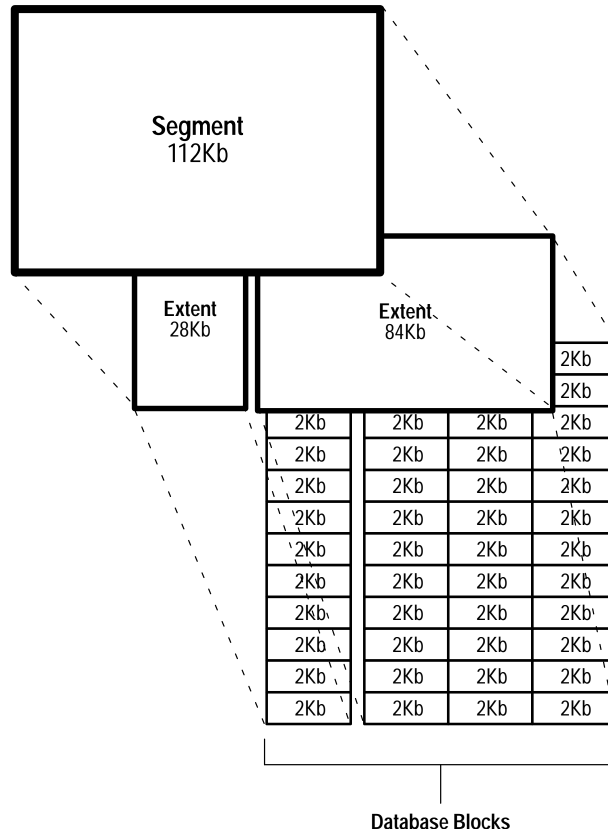


Figure 3 – 1 The Relationship Among Segments, Extents, and Data Blocks

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. You set the data block size for every 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. Oracle data blocks are the smallest units of storage that Oracle can use or allocate.

In contrast, all data at the physical, operating system level is stored in bytes. Each operating system has what is called a *block size*. Oracle requests data in multiples of Oracle blocks, not operating system blocks. Therefore, you should set the Oracle block size to a multiple of the operating system block size to avoid unnecessary I/O.

Extents

The next level of logical database space is called an *extent*. An extent is a specific number of contiguous data blocks that is allocated for storing 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 that have been allocated for a specific type of data structure, and that all 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*.

Oracle allocates space for segments in extents. Therefore, 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. The segments also can span files, but the individual extents cannot.

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.

Data Block Format

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

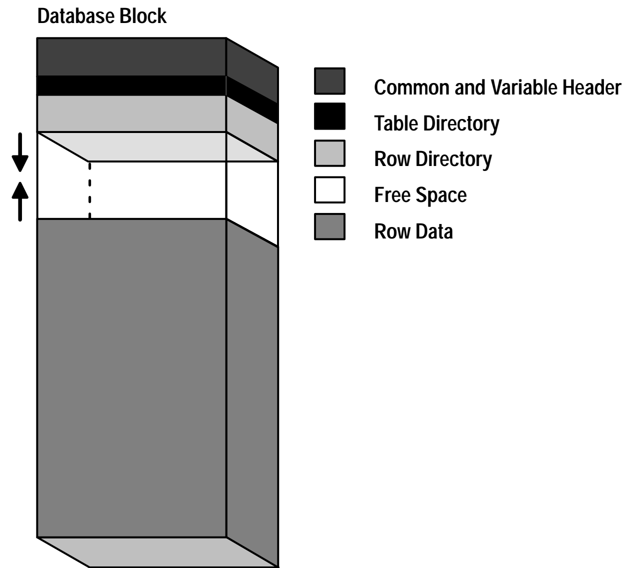


Figure 3 – 2 Data Block Format

Header (Common and Variable)

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

Table Directory

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

Row Directory

This portion of the block contains row 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 block's header, 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 only reuses this space as new rows are inserted in the block.

Row Data

This portion of the block contains table or index data. Rows can span blocks; see "Row Chaining across Data Blocks" on page 3–10.

Free Space

Free space is used for inserting 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 the value for the space management parameter PCTFREE and the amount of current free space in that data block. See “An Introduction to PCTFREE, PCTUSED, and Row Chaining” on page 3–5 for more information on space management parameters.

Space Used for Transaction Entries

Data blocks allocated for the data segment of a table, cluster, or the index segment of an index can also use free space for 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.

An Introduction to PCTFREE, PCTUSED, and Row Chaining

Two space management parameters, PCTFREE and PCTUSED, allow a developer to control the use of free space for inserts of and updates to the rows in data blocks. You specify these parameters only when creating or altering tables or clusters (data segments). You can also specify the storage parameter PCTFREE when creating or altering indexes (index segments).

The PCTFREE Parameter

The PCTFREE parameter is used to set the percentage of a block to be reserved (kept free) for possible updates to rows that already are contained 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 used for this table’s data segment will be kept free and available for possible updates to the existing rows already within each block. Figure 3 – 3 illustrates PCTFREE.

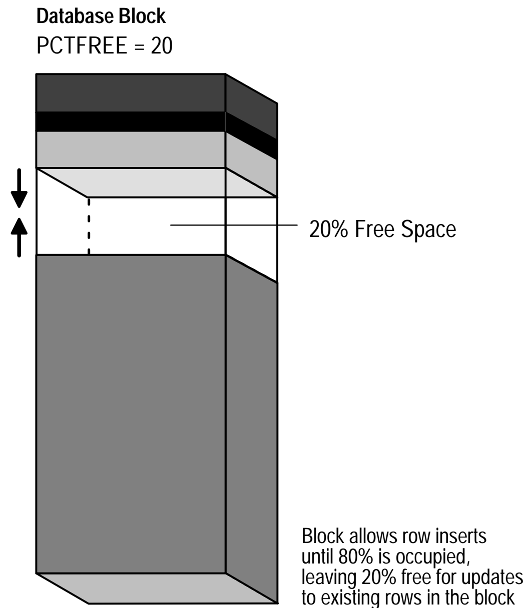


Figure 3 – 3 PCTFREE

Notice that before the block reaches PCTFREE, the free space of the data block is filled by both the insertion of new rows and by the growth of the data block header.

The PCTUSED Parameter

After a data block becomes full, as determined by PCTFREE, Oracle does not consider the block for the insertion of new rows until the percentage of the block being used falls below the parameter PCTUSED. Before 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 within a CREATE TABLE statement:

```
PCTUSED 40
```

In this case, a data block used for this table's data segment is not considered for the insertion of any new rows until the amount of used space in the block falls to 39% or less (assuming that the block's used space has previously reached PCTFREE). Figure 3 – 4 illustrates this.

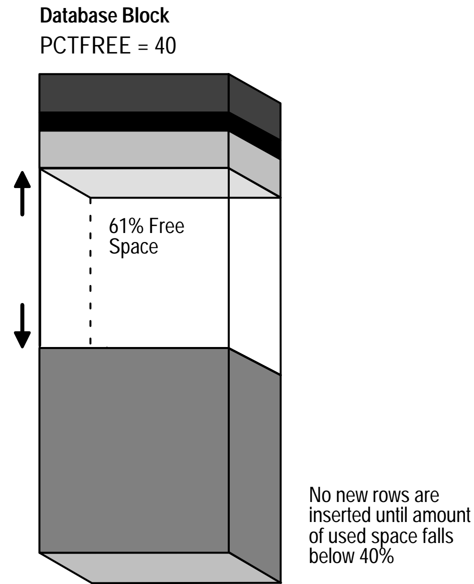
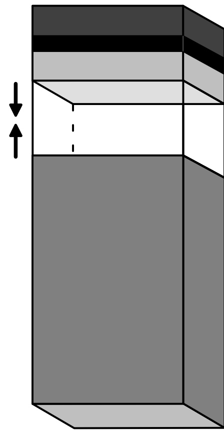


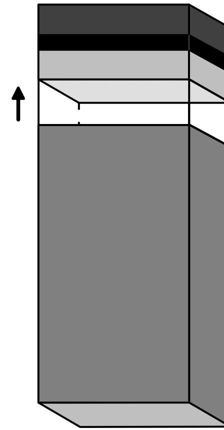
Figure 3 – 4 PCTUSED

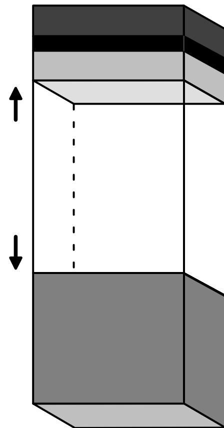
How PCTFREE and PCTUSED Work Together

PCTFREE and PCTUSED work together to optimize the utilization of space in the data blocks of the extents within a data segment. Figure 3 – 5 illustrates how PCTFREE and PCTUSED work together to govern the free space of a data block.

Database Block
PCTFREE = 20, PCTUSED = 40







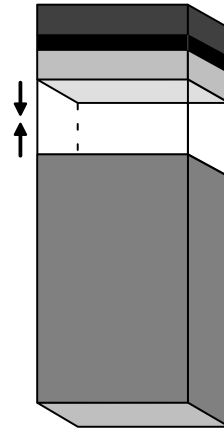


Figure 3 – 5 Maintaining the Free Space of Data Blocks with PCTFREE and PCTUSED

How Oracle Uses 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 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*; a free list is a list 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 block and uses it if possible; if the free space in that block is not large enough to accommodate the INSERT statement, and it 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 DELETE and UPDATE statements, Oracle checks to see if the space being used in the block is less than PCTUSED; if it is, the block goes to the beginning of the free list, and it is the first of the available blocks to be used.

Availability and Compression of Free Space in a Data Block

Two types of statements return space to 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 one transaction and the statement that frees space is in a second transaction (perhaps being executed by another user), the INSERT statement can only use the space made available after the second 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 an INSERT or UPDATE statement attempts to use a block that contains enough free space to contain a new row piece, yet 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 so that the performance of a database system is not decreased by the continuous and unnecessary compression of the free space in data blocks as each DELETE or UPDATE statement is issued.

Row Chaining across Data Blocks

In some circumstances, all of the data for a row in a table may not be able to fit in the same data block. When this occurs, 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 (for example, rows that contain a column of datatype LONG or LONG RAW).

> **Note:** The format of a row and a row piece are described in "Row Format and Size" on page 5–4.

If a table contains a column of datatype LONG, which can hold up to two gigabytes of information, the data for a row may need to be chained to one or more data blocks. Nothing can be done to avoid this type of row chaining.

If a row in a data block is updated so that the overall row length increases and the block's free space has been completely filled, the data for the entire row is *migrated* 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. For information about reducing migrated rows and improving I/O performance, see *Oracle7 Server Tuning.*

Extents

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

This section describes how extents are allocated for segments.

When Extents Are Allocated for Segments

No matter what type, each segment in a database is created with at least one extent to hold its data. This extent is called the segment's *initial* extent.

> **Note:** Rollback segments always have at least two extents.

For example, when you create a table, its data segment contains 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 previous extent in that segment. The next section explains the factors controlling the size of incremental extents.

For maintenance purposes, each segment in a database contains a segment header block that describes the characteristics of that segment and a directory (list) of the extents in that segment.

Extents and the Parallel Query Option

When you use the parallel query option to create indexes and non–clustered tables in parallel, each query 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 three, there will be at least three extents for that index initially.

Serial operations require the object to have at least one extent. Parallel creations require that non–clustered tables or indexes have at least as many extents as there are query servers that create the object.

When you create a table or index with the parallel query option, it is possible to create "pockets" of free space. This occurs when you specify more query servers than there are datafiles in the tablespace. Oracle cannot coalesce this free space with other free space, so this space is available only for subsequent inserts into that table.

For example, if you specify a degree of parallelism of three for a CREATE TABLE ... AS <subquery> statement but there is only one datafile in the tablespace, the situation illustrated in Figure 3 – 6 can arise. Oracle can only coalesce the free space in the last extent of a table or index in each datafile, so all "pockets" of free space within internal table extents of a datafile cannot be coalesced with other free space and allocated as extents.

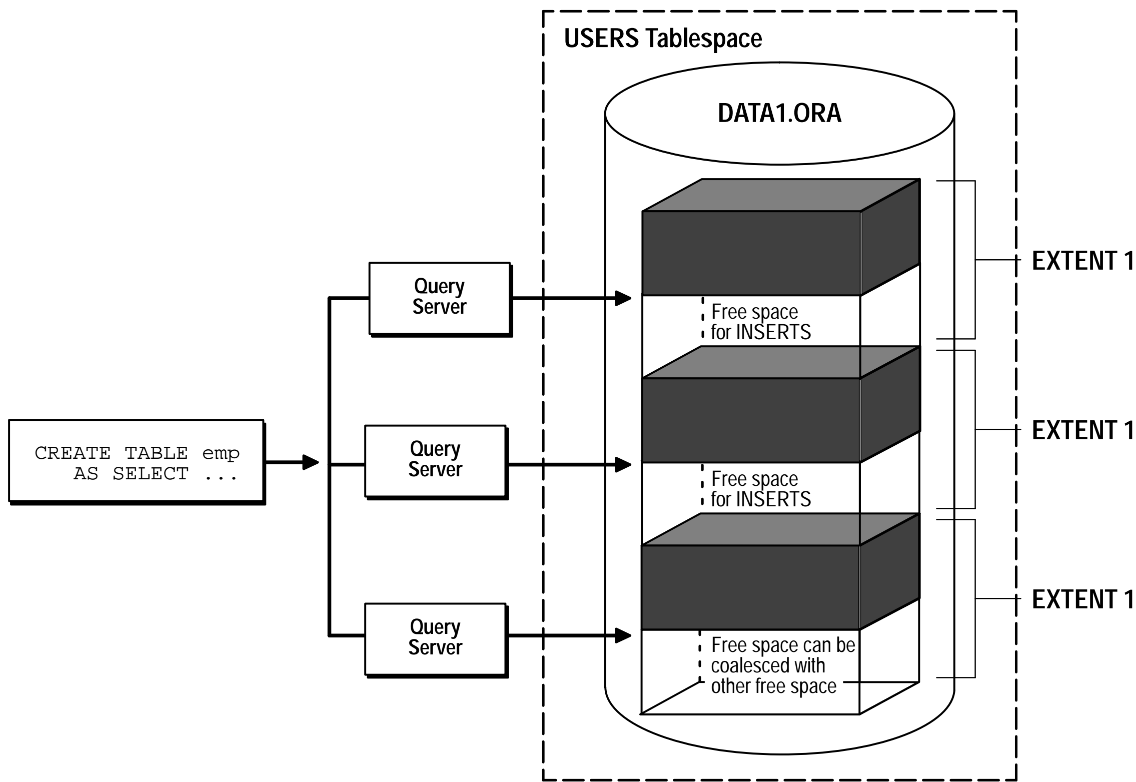


Figure 3 – 6 Unusable Free Space

To alleviate the free space problem, set the degree of parallelism to less than or equal to the number of datafiles in the tablespace that you are placing the non–clustered table or index. Oracle assigns query servers to datafiles in a round–robin fashion, so specifying fewer query servers than datafiles ensures that all free space can be used later by all tables in the tablespace for subsequent extent allocation.

For more information about the parallel query option and creating non–clustered tables and indexes in parallel, see *Oracle7 Server Tuning*.

For more information about datafiles and tablespaces, see Chapter 4, "Tablespaces and Datafiles".

How Extents Are Allocated for Segments

Oracle controls the allocation of extents for a given segment. The procedure to allocate a new extent for a segment is 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. Oracle finds the free space for the new extent by using the following algorithm:

   1.1 Oracle searches for a contiguous set of data blocks that matches the size of new extent, then adds one block to reduce internal fragmentation. For example, if a new extent requires 19 data blocks, Oracle searches for exactly 20 contiguous data blocks. However, if the new extent is 5 or fewer blocks, Oracle does not add an extra block to the request.

   1.2 If an exact match is not found, Oracle then searches for a set of contiguous data blocks equal to or greater than the amount needed. If Oracle finds a group of contiguous blocks that is at least five blocks greater than the size of the extent that is 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 five blocks larger, it allocates all the contiguous blocks.

   Continuing with the example, if Oracle does not find a set of exactly 20 contiguous data blocks, Oracle then searches for a set of contiguous data blocks greater than 20. If the first set that Oracle finds contains 25 or more blocks, it breaks the blocks up and allocates twenty of them to the new extent. Otherwise, it allocates all of the blocks (between 21 and 24) to the new extent.

   1.3 If Oracle does not find a larger set of contiguous data blocks, Oracle then coalesces any free, adjacent data blocks in the corresponding tablespace so that larger sets of contiguous data blocks are formed. (The SMON background process also periodically coalesces adjacent free space.) After coalescing a tablespace's data blocks, Oracle performs the searches described in 1.1. and 1.2.. again. If an extent cannot be allocated after the second search, Oracle returns an error.

2. Once Oracle finds the necessary free space in the tablespace, Oracle allocates a portion of the free space that corresponds to the size of the incremental extent. If Oracle had found a larger amount of free space than was required for the extent, Oracle leaves the remainder as free space (no smaller than five 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.

Usually, Oracle clears the blocks of a newly allocated extent when the extent is first used. In a few cases, however, such as when a database administrator issues an ALTER TABLE or ALTER CLUSTER statement with the ALLOCATE EXTENT option while using free list groups, Oracle clears 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 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.
- A DBA can deallocate unused extents using the following SQL syntax:

```
ALTER TABLE table_name DEALLOCATE UNUSED
```

For More Information on Deallocating Extents

See *Oracle7 Server Administrator's Guide* and *Oracle7 Server SQL Reference.*

When extents are freed, Oracle updates the data dictionary to reflect the regained extents as available space. All data in the blocks of freed extents is inaccessible, and Oracle clears out the data when the blocks are subsequently reused for other extents.

Non–Clustered Tables, Snapshots, and Snapshot Logs

As long as a non–clustered table (including an underlying table for a snapshot or snapshot log) 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.

When you drop a non–clustered table, Oracle reclaims all the extents of its data and index segments for the tablespaces that they were in and makes the extents available for other objects in the tablespace. Subsequently, when other segments require large extents, Oracle identifies and combines contiguous reclaimed extents to form the requested larger extents.

Clustered Tables and Snapshots

Clustered tables and snapshots store their information in the data segment created for the cluster. Therefore, if you drop a clustered table, 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.

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.

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), then Oracle automatically deallocates one or more extents from the rollback segment. See "How Extents Are Deallocated from a Rollback Segment" on page 3–22 for more information.

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.

Determining Sizes and Limits of Segment 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.

Segments

A segment is a set of extents that contain all the data for a specific logical storage structure within a tablespace. For example, for each table, Oracle allocates one or more extents to form that table's data segment, and, for each index, Oracle allocates one or more extents to form its index segment.

There are four types of segments used in Oracle databases:

- data segments
- index segments
- rollback segments
- temporary segments

The following sections discuss each type of segment.

Data Segments

Every non–clustered table (including snapshots and snapshot logs) in an Oracle database has a single data segment to hold all of its data. Oracle creates this data segment when you create the object with the CREATE TABLE/SNAPSHOT/SNAPSHOT LOG command.

Every cluster in an Oracle database uses a single data segment to hold the data for all of its tables. Oracle creates the data segment for the cluster when you issue the CREATE CLUSTER command.

The storage parameters for a table, snapshot, snapshot log, or cluster control the way that its data segment's extents are allocated. You can set these storage parameters directly with the CREATE TABLE/ SNAPSHOT/ SNAPSHOT LOG/CLUSTER or ALTER TABLE/ SNAPSHOT/ SNAPSHOT LOG/CLUSTER commands; these affect the efficiency of data retrieval and storage for the data segment associated with the object. For more information on the various CREATE and ALTER commands, see the *Oracle7 Server SQL Reference*.

Index Segments

Every index in an Oracle database has a single index segment to hold all of its data. Oracle creates the index segment for the index when you issue the CREATE INDEX command. This command allows you to specify the storage parameters for the extents of the index segment and the 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.

Rollback Segments

Each database contains one or more rollback segments. A rollback segment is a portion of the database that records the actions of transactions if the transaction should be rolled back (undone). Rollback segments are used to provide read consistency, to rollback transactions, and to recover the database.

For specific information about how rollback segments function in these situations, see the appropriate sections of this book:

Topic	Section Name	Page
Read Consistency	Multiversion Concurrency Control	10–5
Transaction Rollback	Rolling Back Transactions	12–6
Database Recovery	Rollback Segments and Rolling Back	24–4

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 filename 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 easily be found if necessary for transaction rollback.

Database users or administrators cannot 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.)

Because rollback entries change data blocks, Oracle also records changes to them in the redo log. This second recording of the rollback information is very important for active transactions not yet committed at the time of the 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. Oracle performs rollbacks of transactions that had not been committed or rolled back at the time of the failure after recovery is complete.

When Rollback Information Is Required

Oracle maintains a *transaction table* for each rollback segment contained in a database. Each table is 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 the 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.

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

Transactions and Rollback Segments

Each time a user's transaction begins, Oracle assigns the transaction to a rollback segment:

- 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, a developer or user can specify a particular rollback segment that Oracle should use when executing the transaction. This allows the 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 3 – 7, Figure 3 – 8, and Figure 3 – 9 for more information about how transactions use the extents of a rollback segment.

Each rollback segment can handle a certain 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.

OSDoc

Additional Information: The number of transactions that a rollback segment can handle is an operating system–specific function of the data block size. See your Oracle operating system–specific documentation for more information.

How Extents Are Used and Allocated for Rollback Segments

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.

A transaction writes sequentially to a single rollback segment. Each transaction writes to only one extent of the rollback segment at any given time. Furthermore, many *active* transactions (transactions in progress, not committed or rolled back) can write concurrently to a single rollback segment, even the same extent of a rollback segment; however, each block in a rollback segment's extent can contain information for a single transaction only.

When a transaction runs out of space in the current extent and needs to continue writing, Oracle must find an available extent of the same rollback segment in which to write. Oracle has two options:

- 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 active undo information, Oracle makes it the current extent, and all transactions that need more space from then on can write rollback information to the new current extent. Figure 3 – 7 illustrates two transactions, T1 and T2, which continue writing from the third extent to the fourth extent of a rollback segment.

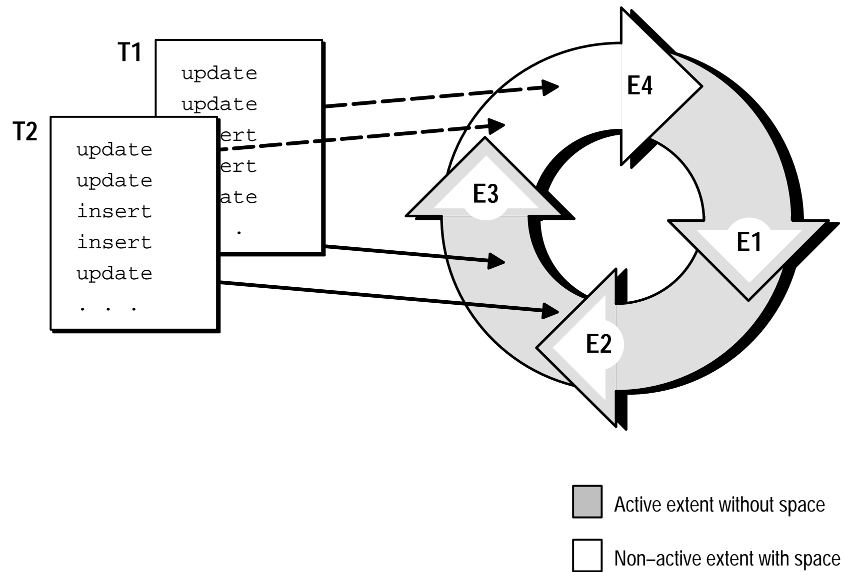


Figure 3 – 7 Use of Allocated Extents in a Rollback Segment

As the transactions continue writing and fill the current extent, Oracle checks the next extent already allocated for the rollback segment to determine if it is available. In Figure 3 – 8, 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 this extent. This figure shows the cyclical nature of extent use in rollback segments.

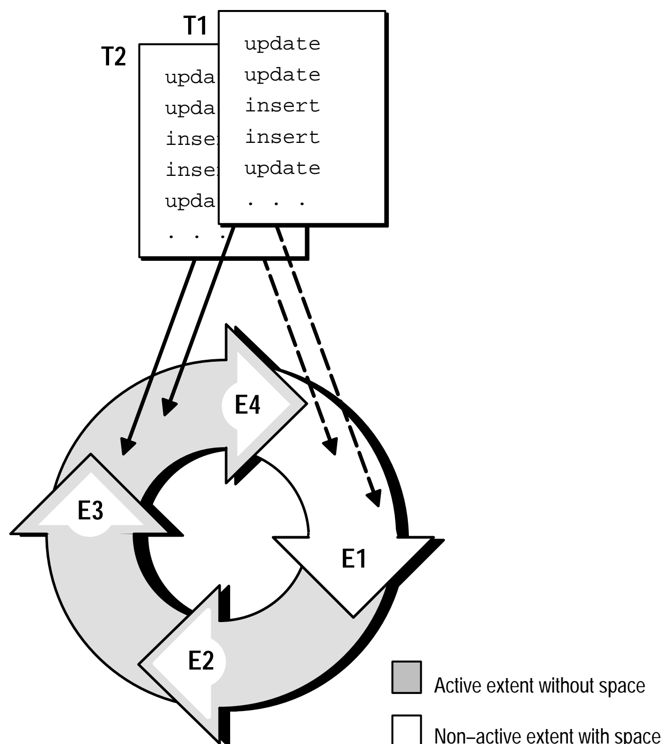


Figure 3 – 8 Cyclical Use of the Allocated Extents in a Rollback Segment

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 active data, then Oracle must allocate a new extent. Oracle can allocate new extents for a rollback segment until the number of extents reaches the value set for the rollback segment's storage parameter MAXEXTENTS.

Figure 3 – 9 shows when a new extent must be allocated for a rollback segment. The uncommitted transactions are long running (either idle, active, or persistent in–doubt distributed transactions). At this time, they are writing to the fourth extent, E4, in the rollback segment. However, when E4 is completely full, the transactions cannot continue further writing to the next extent in sequence, E1, because it contains active rollback entries. Therefore, Oracle allocates a new extent, E5, for this rollback segment, and the transactions continue writing to this new extent.

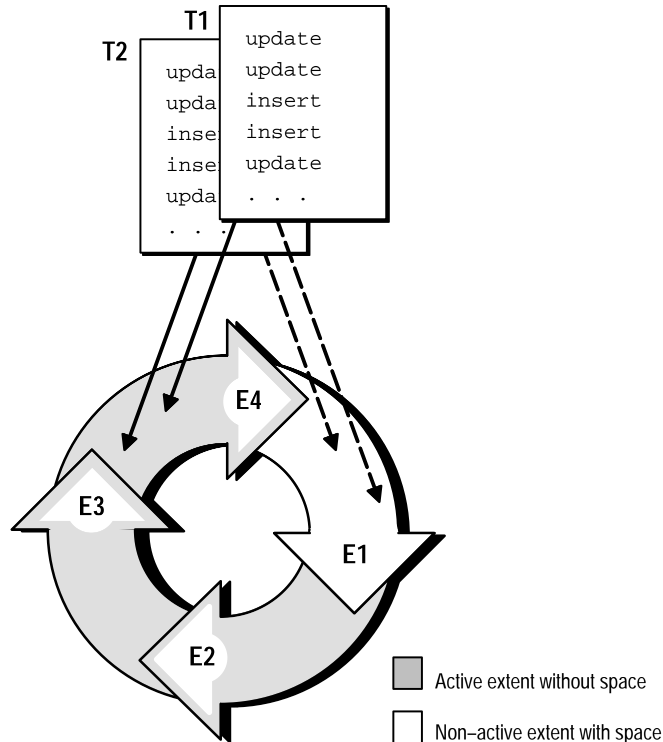


Figure 3 – 9 Allocation of a New Extent for a Rollback Segment

How Extents Are Deallocated from a Rollback Segment

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 rollback 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 the extents immediately following the extent just filled are inactive, Oracle deallocates consecutive non–active extents from the rollback segment until the total size of the rollback segment is equal to or as 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:

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

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.

Instances and Types of Rollback Segments

When an 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:

- 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, 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.
- The instance first tries to acquire all private rollback segments specified by the instance's ROLLBACK_SEGMENTS parameter. If one instance 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.
- 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.
- If the instance already has acquired enough private rollback segments in Step 2, no further action is required. However, if an instance requires more rollback segments, the instance attempts to acquire public rollback segments. (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.

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.

Note: 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 (see the following section). However, when using the Oracle Parallel Server, you may want to use private rollback segments. See *Oracle7 Parallel Server Concepts & Administration* for more information about rollback segment use in an Oracle Parallel Server.

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.

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 the following:

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).

Figure 3 – 10 shows how a rollback segment moves from one state to another.

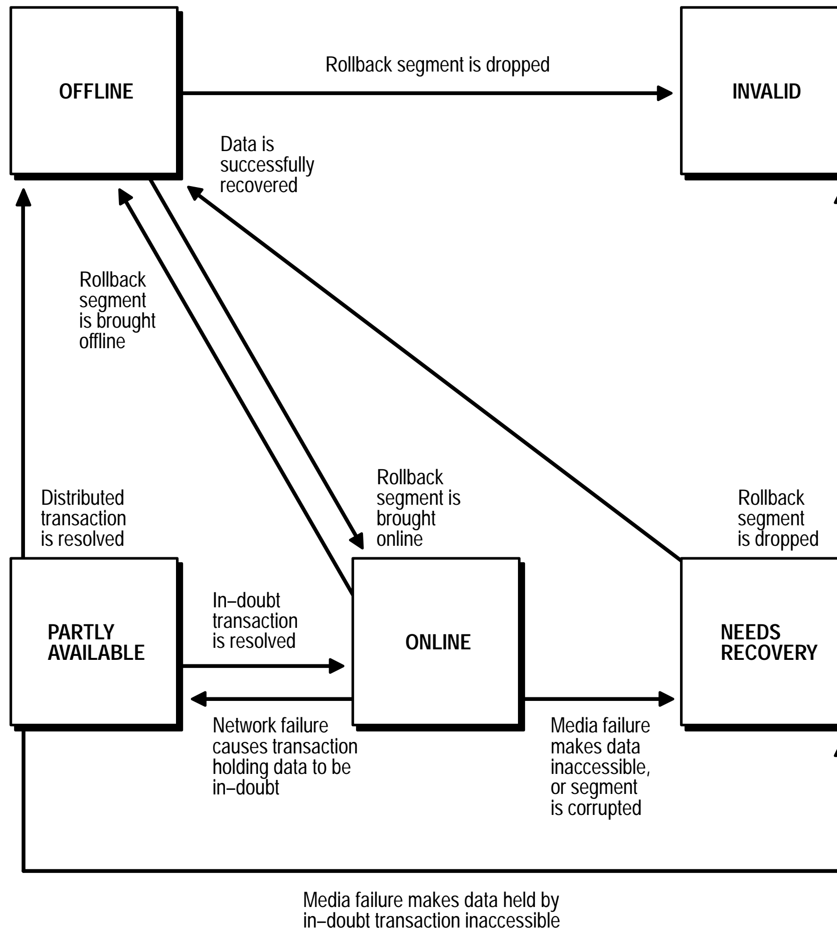


Figure 3 – 10 Rollback Segment States and State Transitions

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. The differences between the two states are the following:

- 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. (See *Oracle7 Server Distributed Systems, Volume I* for information about failures in distributed transactions.)

If you bring a PARTLY AVAILABLE rollback segment online (by a command 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. (See “When Rollback Information Is Required” on page 3–17 for information on the transaction table.)

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.

Viewing the State of a Rollback Segment The data dictionary table DBA_ROLLBACK_SEGS lists the status (state) of each rollback segment, along with other rollback segment information.

Deferred Rollback Segments

When a tablespace goes offline such that transactions cannot be rolled back immediately, Oracle writes a *deferred rollback segment*. The deferred rollback segment contains the rollback entries that could not be applied to the tablespace, so 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.

Temporary Segments

When processing queries, Oracle often requires temporary workspace for intermediate stages of SQL statement processing. 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 commands may require the use of a temporary segment:

- CREATE INDEX
- SELECT ... ORDER BY
- SELECT DISTINCT ...
- SELECT ... GROUP BY
- SELECT ... UNION
- SELECT ... INTERSECT
- SELECT ... MINUS
- unindexed joins
- certain correlated subqueries

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. For more information on SORT_AREA_SIZE and other initialization parameters, see the *Oracle7 Server Reference.*

How Temporary Segments Are Allocated

Oracle allocates temporary segments as needed during a user session. For example, a user might issue a query that requires three temporary segments. Oracle drops temporary segments when the statement completes. The default storage characteristics of the containing tablespace determine those of the extents of the temporary segment.

Oracle creates temporary segments in the temporary tablespace of the user issuing the statement. You specify this tablespace with a CREATE USER or an ALTER USER command using the TEMPORARY TABLESPACE option. Otherwise, the default temporary tablespace is the SYSTEM tablespace. For more information about assigning a user's temporary segment tablespace, see Chapter 17, "Database Access".

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.

The redo log does not contain entries for changes to temporary segments used for sort operations.

CHAPTER

4

Tablespaces and Datafiles

Space — the final frontier...

Gene Roddenberry: *Star Trek*

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

- An Introduction to Tablespaces and Datafiles
- Tablespaces
- Datafiles

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for more information about tablespaces and datafiles in that environment.

An Introduction to Tablespaces and Datafiles

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

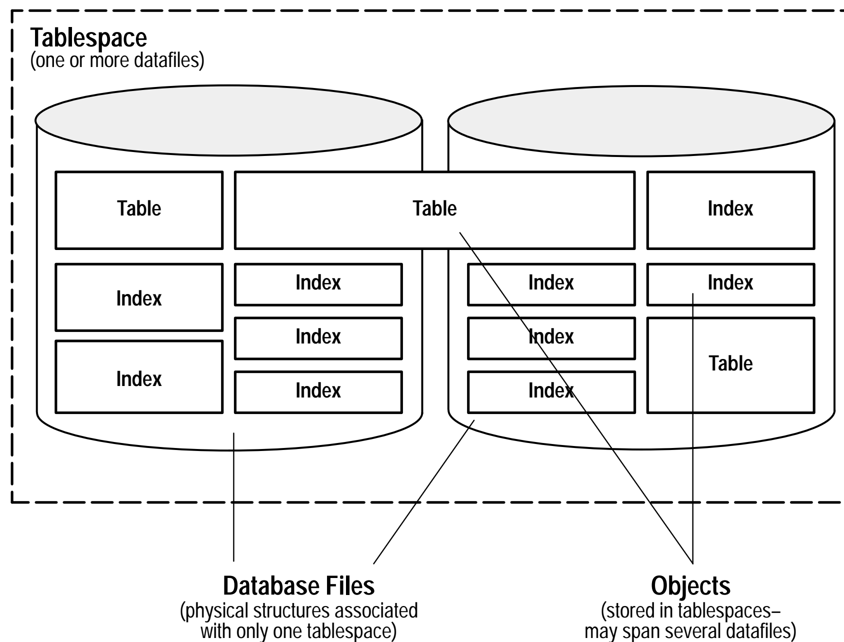


Figure 4 – 1 Datafiles and Tablespaces

Although databases, tablespaces, datafiles, and segments are closely related, they have important differences:

databases and tablespaces	An Oracle database is comprised of one or more logical storage units called *tablespaces*. The database's data is collectively stored in the database's tablespaces.
tablespaces and datafiles	Each tablespace in an Oracle database is comprised of one or more operating system files called *datafiles*. A tablespace's datafiles physically store the associated database data on disk.

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. A more complicated database might have three tablespaces, each comprised of two datafiles (for a total of six datafiles).
schema objects, segments, and tablespaces	When a schema object such as a table or index is created, its segment is created within a designated tablespace in the database. For example, suppose you create a table in a specific tablespace using the CREATE TABLE command with the TABLESPACE option. Oracle allocates the space for this table's data segment in one or more of the datafiles that constitute the specified tablespace. An object's segment allocates space in only one tablespace of a database. See Chapter 3, "Data Blocks, Extents, and Segments", for more information about extents and segments and how they relate to tablespaces.

The following sections further explain tablespaces and datafiles.

Tablespaces

A database is divided into one or more logical storage units called *tablespaces.* A database administrator can use tablespaces to do the following:

- 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 can create new tablespaces, add and remove datafiles from tablespaces, set and alter default segment storage settings for segments created in a tablespace, make a tablespace read–only or writeable, make a tablespace temporary or permanent, and drop tablespaces.

This section includes the following topics:

- The SYSTEM Tablespace
- Allocating More Space for a Database
- Online and Offline Tablespaces
- Read–Only Tablespaces
- Temporary Tablespaces

The SYSTEM Tablespace

Every Oracle database contains a tablespace named SYSTEM that Oracle creates automatically when the database is created. The SYSTEM tablespace always contains the data dictionary tables for the entire database.

A small database might need only the SYSTEM tablespace; however, it is recommended that you create at least one additional tablespace to store user data separate from data dictionary information. This allows you more flexibility in various database administration operations and can reduce contention among dictionary objects and schema objects for the same datafiles.

> **Note:** The SYSTEM tablespace must always be kept online. See "Online and Offline Tablespaces" on page 4–6.

All data stored on behalf of stored PL/SQL program units (procedures, functions, packages and triggers) resides in the SYSTEM tablespace. If you create many of these PL/SQL objects, the database administrator needs to plan for the space in the SYSTEM tablespace that these objects use. For more information about these objects and the space that they require, see Chapter 14, "Procedures and Packages", and Chapter 15, "Database Triggers".

Allocating More Space for a Database

To enlarge a database, you have three options. You can add another datafile to one of its existing tablespaces, thereby increasing the amount of disk space allocated for the corresponding tablespace. Figure 4 – 2 illustrates this kind of space increase.

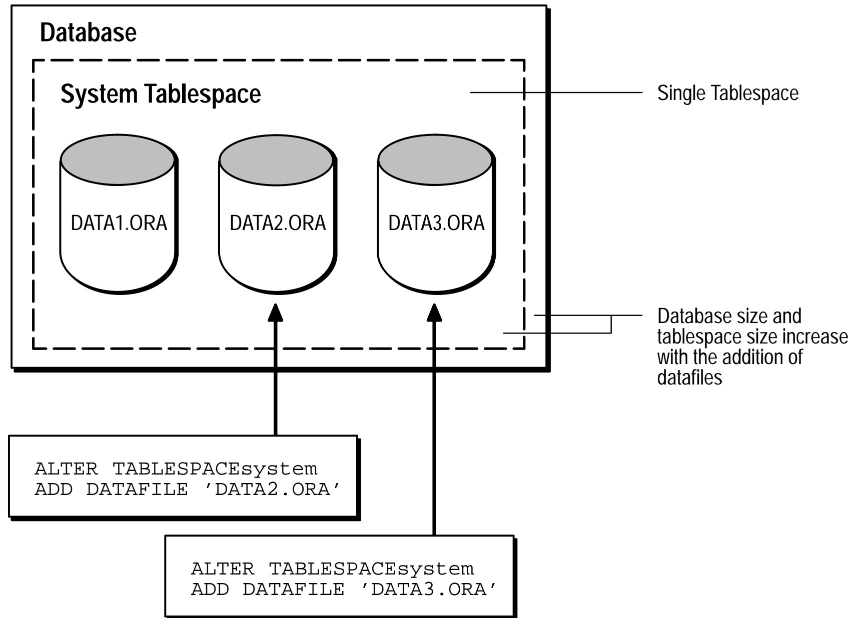


Figure 4 – 2 Enlarging a Database by Adding a Datafile to a Tablespace

Alternatively, a database administrator can create a new tablespace (defined by an additional datafile) to increase the size of a database. Figure 4 – 3 illustrates this.

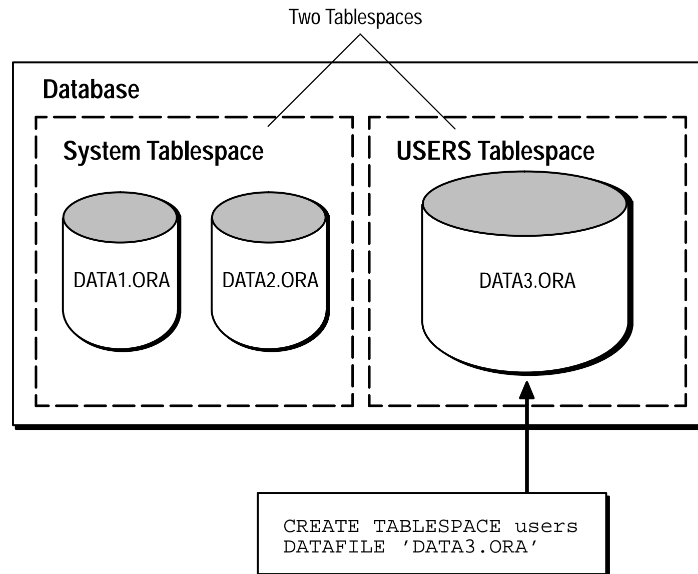


Figure 4 – 3 Enlarging a Database by Adding a New Tablespace

The size of a tablespace is the size of the datafile(s) that constitute the tablespace, and the size of a database is the collective size of the tablespaces that constitute the database.

The third option 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 4 – 4 illustrates this.

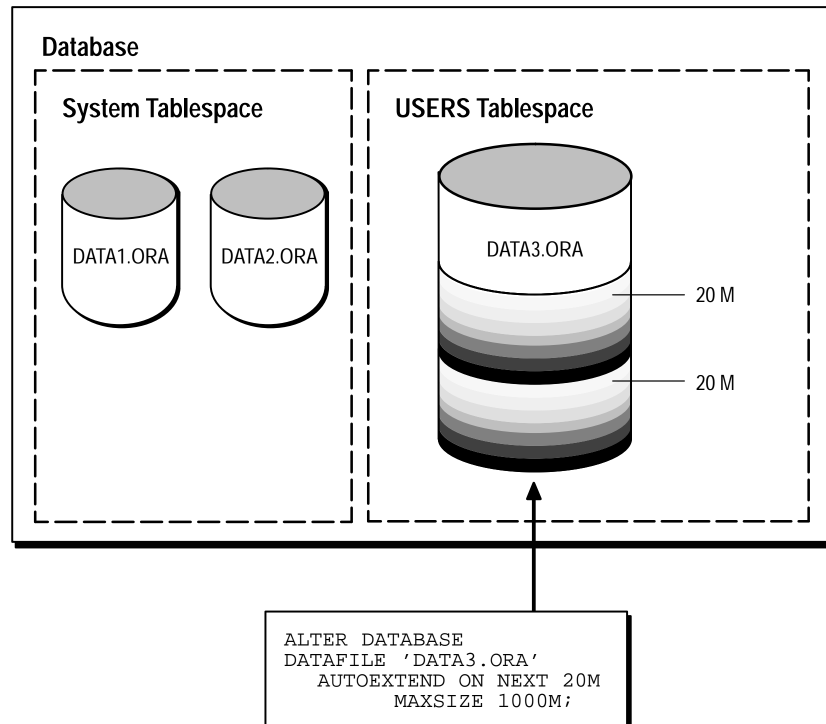


Figure 4 – 4 Enlarging a Database by Dynamically Sizing Datafiles

For more information about increasing the amount of space in your database, see the *Oracle7 Server Administrator's Guide*.

Online and Offline Tablespaces

A database administrator can bring any tablespace (except the SYSTEM tablespace) in an Oracle database *online* (accessible) or *offline* (not accessible) whenever the database is open.

> **Note:** The SYSTEM tablespace must always be online 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 for any of the following reasons:

- 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

When a Tablespace Goes Offline

When a tablespace goes offline, Oracle does not permit any subsequent SQL statements to reference objects contained in the tablespace. Active transactions with completed statements that refer to data in a tablespace that has been taken offline are not affected at the transaction level. Oracle saves rollback data corresponding to statements that affect data in the offline tablespace 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.

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

When a tablespace goes offline or comes back online, it 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, tablespaces cannot be transferred from database to database (transfer of Oracle data can be achieved with tools described in *Oracle7 Server Utilities)*.

Oracle automatically changes a tablespace from online to offline when certain errors are encountered (for example, when the database writer process, DBWR, fails in several attempts to write to a datafile of the tablespace). Users trying to access tables in the tablespace with the problem receive an error. If the problem that causes this disk I/O to fail is media failure, the tablespace must be recovered after you correct the hardware problem.

Using Tablespaces for Special Procedures

By using multiple tablespaces to separate different types of data, the database administrator can also take specific tablespaces offline for certain procedures, while 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 determines that it 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. The READ ONLY option of the ALTER TABLESPACE command allows you to change the tablespace to read–only, making all of its associated datafiles read–only as well. You can then use the READ WRITE option to make a read–only tablespace writeable again.

Read–only tablespaces cannot be modified. Therefore, 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.

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.

Making a Tablespace Read–Only

Use the SQL command ALTER TABLESPACE to change a tablespace to read–only. For information on the ALTER TABLESPACE command, see the *Oracle7 Server SQL Reference.*

Read–Only vs. Online or Offline

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 readable. The file cannot be written to unless its associated tablespace is returned to the read–write state. The files of a read–only tablespace can independently be taken online or offline using the DATAFILE option of the ALTER DATABASE command.

Restrictions on Read–Only Tablespaces

You cannot add datafiles to a tablespace that is read–only, 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.

To update a read–only tablespace, you must first make the tablespace writeable. After updating the tablespace, you can then reset it to be read–only.

Read–Only Tablespaces and Recovery

Read–only tablespaces have several implications upon instance or media recovery. See Chapter 24, “Database Recovery”, for more information about recovery.

Temporary Tablespaces

Space management for sort operations is performed more efficiently using temporary tablespaces designated exclusively for sorts. This scheme 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 command to collect optimizer statistics, benefit from temporary tablespaces. The performance gains are significant in parallel server environments.

A *temporary tablespace* is a tablespace that can only be used for sort segments. No permanent objects can reside in a temporary tablespace. Sort segments are used when a segment is shared by multiple sort operations. One sort segment exists in 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 grows 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.

You create temporary tablespaces using the following SQL syntax:

```
CREATE TABLESPACE tablespace TEMPORARY
```

You can also alter a tablespace from PERMANENT to TEMPORARY or vice versa using the following syntax:

```
ALTER TABLESPACE tablespace TEMPORARY
```

For more information on the CREATE TABLESPACE and ALTER TABLESPACE Commands, see Chapter 4 of *Oracle7 Server SQL Reference.*

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.

When a datafile is created for a tablespace, Oracle creates the file by allocating the specified amount of disk space plus the overhead required for the file header. When a datafile is created, the operating system 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.

OSDoc

Additional Information: For information on the amount of space required for the file header of datafiles on your operating system, see your Oracle operating system specific documentation.

Since the first tablespace in any database is always the SYSTEM tablespace, Oracle automatically allocates the first datafiles of any database for the SYSTEM tablespace during database creation.

Datafile Contents

After a datafile is initially created, the allocated disk space does not contain any data; however, Oracle reserves the space to hold only the data for future segments of the associated tablespace — it cannot store any other program's data. As a segment (such as the data segment for a table) is created and grows in a tablespace, Oracle uses the free space in the associated datafiles to allocate extents for the segment.

The data in the segments of objects (data segments, index segments, rollback segments, and so on) in a tablespace are 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 object within a specific tablespace. Oracle allocates the extents of a single segment in one or more datafiles of a tablespace; therefore, an object can "span" one or more datafiles. Unless table "striping" is used, the database administrator and end–users cannot control which datafile stores an object.

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 objects in the tablespace grow. This functionality allows you to have fewer datafiles per tablespace and can simplify administration of datafiles.

For more information about resizing datafiles, see the *Oracle7 Server Administrator's Guide*.

Offline Datafiles

You can take tablespaces *offline* (make unavailable) or bring them *online* (make available) at any time. Therefore, all 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 certain database recovery procedures.

CHAPTER

5

Schema Objects

My object all sublime
I shall achieve in time —
To let the punishment fit the crime.

Sir William Schwenck Gilbert: *The Mikado*

This chapter discusses the different types of objects contained in a user's schema. It includes:

- Overview of Schema Objects
- Tables
- Views
- The Sequence Generator
- Synonyms
- Indexes
- Clusters
- Hash Clusters

Certain kinds of schema objects are discussed in more detail elsewhere in this manual. Specifically, procedures, functions, and packages are discussed in Chapter 14, "Procedures and Packages", database triggers in Chapter 15, "Database Triggers, and snapshots are covered in Chapter 21, "Distributed Databases".

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for additional information about schema objects in that environment.

Overview of Schema Objects

Associated with each database user is a *schema*. A schema is a collection of schema objects. Examples of schema objects include tables, views, sequences, synonyms, indexes, clusters, database links, procedures, and packages. This chapter explains tables, views, sequences, synonyms, indexes, and clusters.

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. Figure 5 – 1 illustrates the relationship among objects, tablespaces, and datafiles.

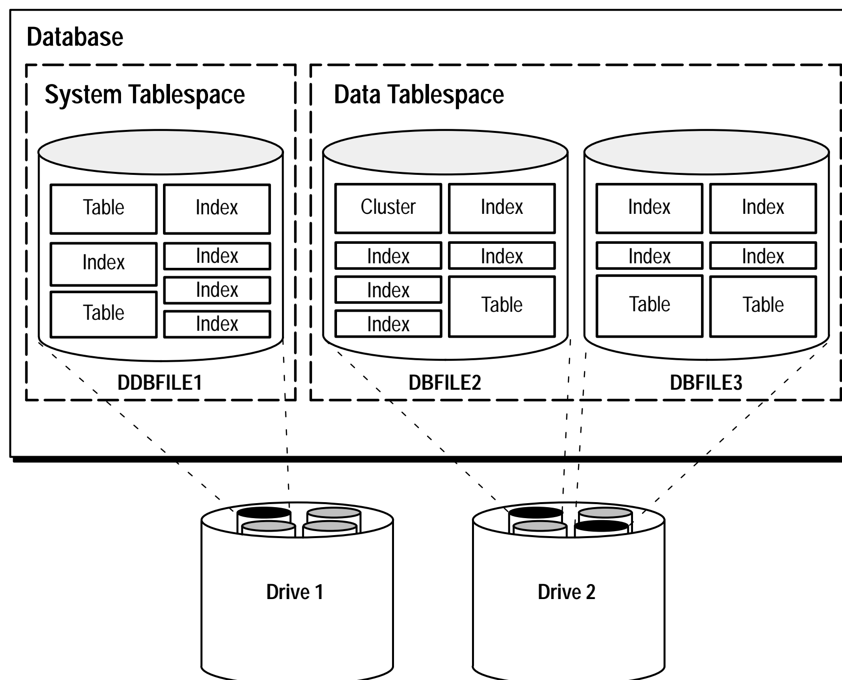


Figure 5 – 1 Schema Objects, Tablespaces, and Datafiles

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.

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) or *precision* and *scale* (for columns of the NUMBER datatype only). A row is a collection of column information corresponding to a single record.

> **Note:** See Chapter 6, “Datatypes”, for a discussion of the Oracle datatypes.

You can optionally specify rules for each column of a table. These rules are called *integrity constraints*. One example is a NOT NULL integrity constraint. This constraint forces the column to contain a value in every row. See Chapter 7, “Data Integrity”, for more information about integrity constraints.

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

Figure 5 – 2 shows a table named EMP.

Rows
Columns
Column names

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CLERK	7902	17–DEC–88	800.00	300.00	20
7499	ALLEN	SALESMAN	7698	20–FEB–88	1600.00	300.00	30
7521	WARD	SALESMAN	7698	22–FEB–88	1250.00	500.00	30
7566	JONES	MANAGER	7839	02–APR–88	2975.00		20

Column not allowing nulls
Column allowing nulls

Figure 5 – 2 The EMP Table

How Table Data Is Stored

When you create a non–clustered 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. 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 non–clustered table's data segment is either the table owner's default tablespace or a tablespace specifically named in the CREATE TABLE statement. See "User Tablespace Settings and Quotas" on page 17–6.

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, 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. See "Row Chaining across Data Blocks" on page 3–10.

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 5 – 3 shows the format of a row piece.

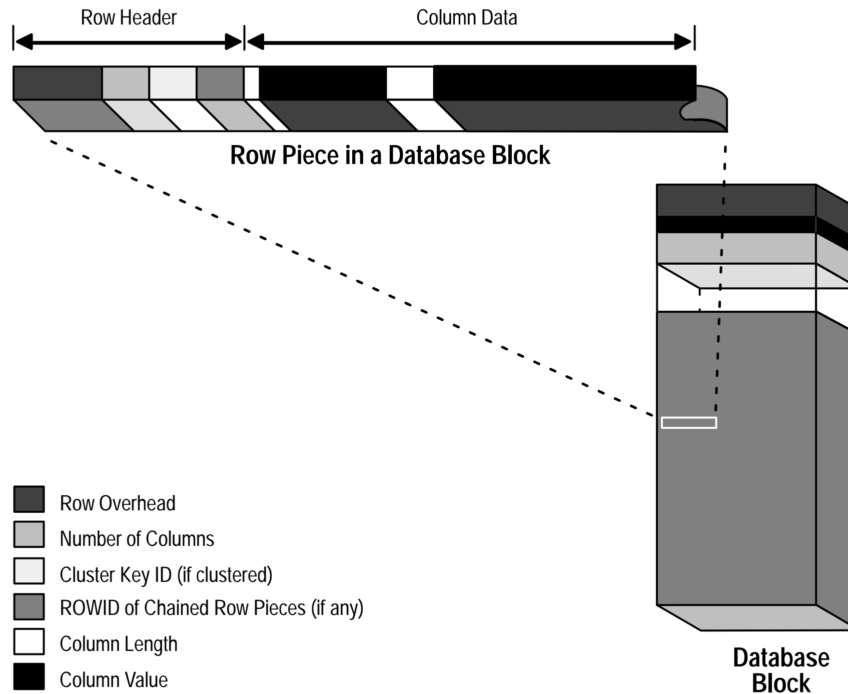


Figure 5 – 3 The Format of a Row Piece

The *row header* precedes the data and contains information about

- row pieces
- (for chained row pieces) chaining
- columns in the row piece
- (for clustered data) cluster keys

A non–clustered 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, 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 store the column length because the row header signals the start of a new row (for example, the last three columns of a table are null, thus there is no information stored for those columns).

> **Note:** Each row uses two bytes in the data block header's row directory.

Clustered rows contain the same information as non–clustered rows. In addition, they contain information that references the cluster key to which they belong. See "Clusters" on page 5–23.

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, Oracle always stores this column last. Also, if a table is altered so that a new column is added, 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, the benefits of placing frequently null columns last are lost.

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 IMPORT and EXPORT utilities. If the cluster key values of a row change, 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 "ROWIDs and the ROWID Datatype" on page 6–9.

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. 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. For more information about comparisons using IS NULL and the NVL function, see *Oracle7 Server SQL Reference*.

Nulls are not indexed, except when the cluster key column value is null.

Default Values for Columns

You can assign a column of a table a default value 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 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, the default for the column is implicitly set to NULL.

When Default Values Are Inserted Relative to Integrity Constraint Checking

Integrity constraint checking occurs after the row with a default value is inserted. For example, in Figure 5 – 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 employee's department number, the DEPTNO column's default value "20" is supplied. After the default value is supplied, the FOREIGN KEY integrity constraint defined on the DEPTNO column is checked.

Parent Key

Table DEPT

DEPNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO

Foreign Key

Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9000.00		20
7499	ALLEN	VP_SALES	7329	20–FEB–90	7500.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	5000.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90	2975.00	400.00	30
7691	OSTER	SALESMAN	7521	06–APR–90	2975.00	400.00	20

Default Value
(if no value is given for this column, the default of 20 is used)

New row to be inserted, without value for DEPTNO column.

INSERT INTO

7691	OSTER	SALESMAN	7521	06–APR–90	2975.00	400.00

Figure 5 – 4 DEFAULT Column Values

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 only want users to see five of these columns, or only specific rows, you can create a view of that table for other users to access. Figure 5 – 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.

Base Table

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CLERK	7902	17–DEC–88	300.00	800.00	20
7499	ALLEN	SALESMAN	7698	20–FEB–88	300.00	1600.00	30
7521	WARD	SALESMAN	7698	22–FEB–88	5.00	1250.00	30
7566	JONES	MANAGER	7839	02–APR–88		2975.00	20

View

STAFF

EMPNO	ENAME	JOB	MGR	DEPTNO
7329	SMITH	CLERK	7902	20
7499	ALLEN	SALESMAN	7698	30
7521	WARD	SALESMAN	7698	30
7566	JONES	MANAGER	7839	20

Figure 5 – 5 An Example of a View

Since views are derived from tables, many similarities exist between the two. For example, you can define views with up to 254 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.

For More Information

See *Oracle7 Server 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 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 and/or columns of a table

  For example, Figure 5 – 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 commands 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, 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. For information about GROUP BY or UNION, see the *Oracle7 Server SQL Reference*.

- to save complex queries

  For example, a query could perform extensive calculations with table information. By saving this query as a view, the calculations can be performed each time the view is queried.

- to achieve improvements in availability and performance

  For example, a database administrator can divide a large table into smaller tables (partitions) for many reasons, including partition level

load, purge, backup, restore, reorganization, and index building. Once partition views are defined, users can query partitions, rather than very large tables. This ability to prune unneeded partitions from queries increases performance and availability.

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 merges the statement that references the view with the query that defines the view and then parses the merged statement in a shared SQL area and executes it. Oracle parses a statement that references a view in a new shared SQL area **only** if no existing shared SQL area contains an identical statement. Therefore, you obtain the benefit of reduced memory usage associated with shared SQL when you use views.

NLS Parameters

In evaluating 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.

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 the 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 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.

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 re–create it, Oracle determines whether the new base table is acceptable to the existing definition of the view. See Chapter 16, "Dependencies Among Schema Objects", for a complete discussion of dependencies in a database.

Updatable Join Views

A *join view* is defined as a view with more than one table or view in its FROM clause and which 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, which involves two or more base tables or views, where UPDATE, INSERT, and DELETE operations are permitted. The data dictionary views, ALL_UPDATABLE_COLUMNS, DBA_UPDATABLE_COLUMNS, and USER_UPDATABLE_COLUMNS, contain information that indicates which of the view columns are updatable.

Table 5 – 1 lists rules for updatable 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 non–updatable.
DELETE Rule	Rows from a join view can be deleted as long as 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.

Table 5 – 1 Rules for INSERT, UPDATE, and DELETE on Join Views

Partition Views

The database administrator can use partition views to divide a very large table into multiple smaller pieces (or partitions) to achieve significant improvements in availability, administration and performance. The basic idea behind partition views is simple: divide the large table into multiple physical tables using a partitioning criteria; glue the partitions together into a whole for query purposes. A partition view can assign key ranges to partitions. Queries that use a key range to select from a partitions view will access only the partitions that lie within the key range.

For example, sales data for a calendar year may be broken up into four separate tables, one per quarter: Q1_SALES, Q2_SALES, Q3_SALES and Q4_SALES.

Partition Views Using Check Constraints

A partition view may then be defined by using check constraints or by using WHERE clauses. Here is the preferred method that uses check constraints:

```
ALTER TABLE Q1_SALES ADD CONSTRAINT C0 check (sale_date between
  'jan-1-1995' and 'mar-31-1995');
ALTER TABLE Q2_SALES ADD CONSTRAINT C1 check (sale_date between
  'apr-1-1995' and 'jun-30-1995');
ALTER TABLE Q3_SALES ADD CONSTRAINT C2 check (sale_date between
  'jul-1-1995' and 'sep-30-1995');
ALTER TABLE Q4_SALES ADD CONSTRAINT C3 check (sale_date between
  'oct-1-1995' and 'dec-31-1995');
CREATE VIEW sales AS
  SELECT * FROM Q1_SALES UNION ALL
  SELECT * FROM Q2_SALES UNION ALL
  SELECT * FROM Q3_SALES UNION ALL
  SELECT * FROM Q4_SALES;
```

This method has several advantages. The check constraint predicates are not evaluated per row for queries. The predicates guard against inserting rows in the wrong partitions. It is easier to query the dictionary and find the partitioning criteria.

Partition Views Using WHERE Clauses

Alternatively, you can express the criteria in the WHERE clause of a view definition:

```
CREATE VIEW sales AS
  SELECT * FROM Q1_SALES WHERE sale_date between
  'jan-1-1995' and 'mar-31-1995' UNION ALL
  SELECT * FROM Q2_SALES WHERE sale_date between
  'apr-1-1995' and 'jun-30-1995' UNION ALL
  SELECT * FROM Q3_SALES WHERE sale_date between
  'jul-1-1995' and 'sep-30-1995' UNION ALL
  SELECT * FROM Q4_SALES WHERE sale_date between
  'oct-1-1995' and 'dec-31-1995';
```

This method has several drawbacks. First, the partitioning predicate is applied at runtime for all rows in all partitions that are not skipped. Second, if the user mistakenly inserts a row with sale_date = 'apr–4–1995' in Q1_SALES, the row will "disappear" from the partition view. Finally, the partitioning criteria are difficult to retrieve from the data dictionary because they are all embedded in one long view definition.

However, using WHERE clauses to define partition views has one advantage over using check constraints: the partition can be on a remote database with WHERE clauses. For example, you can use a WHERE clause to define a partition on a remote database as in this example:

```
SELECT * FROM eastern_sales@east.acme.com WHERE LOC = 'EAST'
  UNION ALL
  SELECT * FROM western_sales@west.acme.com WHERE LOC = 'WEST';
```

Because queries against eastern sales data do not need to fetch any western data, users will get increased performance. This cannot be done with constraints because the distributed query facility does not retrieve check constrains from remote databases.

Benefits of Partition Views

Partition views 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. Because the partitions are independent of each other, unavailability of a piece (or a subset of pieces) does not affect access to the rest of the data. The Oracle server incorporates the intelligence to explicitly recognize partition views. This knowledge is exploited in query optimization and query execution in several ways:

- partition elimination

  For each query, depending on the selection criteria specified, unneeded partitions can be eliminated. For example, if a query only involves Q1 sales data, there is no need to retrieve data for the remaining three quarters. Such intelligent elimination can drastically reduce the data volume, resulting in substantial improvements in query performance.

- partition–level query optimization

  Query execution is optimized at the level of underlying physical tables, selecting the most appropriate access path for each piece based on the amount of data to be examined. Consider an example of a partition view ORDERS consisting of 12 partitions, one for each month: ORDERS_JAN, ORDERS_FEB, ..., ORDERS_DEC. Consider the following query against this view:

  ```
  SELECT orderno, value, custno FROM orders
    WHERE order_date BETWEEN '30–JAN–95' AND '25–FEB–95';
  ```

  This query involves just a few days of data for ORDERS_JAN and most of the data for ORDERS_FEB. Given this, the optimizer may come up with a plan that uses indexed access of ORDERS_JAN and a

full scan of the table ORDERS_FEB. Examination of the remaining 10 partitions will be eliminated since the query does not involve them.

Partition views are especially useful in data warehouse environments where there is a common need to store and analyze large amounts of historical data.

For More Information See *Oracle7 Server Tuning.*

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 it ascends or descends, the interval between numbers, and other information. One important part of a sequence's definition is 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. Once 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 one or for multiple tables. 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.

For more performance implications when using sequences, see the *Oracle7 Server Application Developer's Guide*.

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 contained in the schema of a specific user who has control over its availability to others.

Synonyms are very useful in both distributed and non–distributed 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, only the synonym needs to be redefined and 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:

- There is a table called SALES_DATA, contained in the schema owned by the user named JWARD.
- The SELECT privilege for the SALES_DATA table is granted to PUBLIC.

At this point, you would 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 explicitly to speed SQL statement execution on a 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.

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 anytime without effecting 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 once 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.

Unique and Non–Unique Indexes

Indexes can be unique or non–unique. Unique indexes guarantee that no two rows of a table have duplicate values in the columns that define the index. Non–unique 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, you should give some thought to the order of the columns used in the definition; generally, the most commonly accessed or most selective columns go first. For more information on composite indexes, see *Oracle7 Server Tuning*.

Figure 5 – 6 illustrates the VENDOR_PARTS table that has a composite index on the VENDOR_ID and PART_NO columns.

VENDOR_PARTS

VEND ID	PART NO	UNIT COST
1012	10–440	.25
1012	10–441	.39
1012	457	4.95
1010	10–440	.27
1010	457	5.10
1220	08–300	1.33
1012	08–300	1.19
1292	457	5.28

Concatenated Index
(index with multiple columns)

Figure 5 – 6 Indexes, Primary keys, Unique Keys, and Foreign Keys

No more than 16 columns can form the composite index, and a key value cannot exceed roughly one–half (minus some overhead) the available data space in a data block.

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.

Integrity constraints enforce the business rules of a database; see Chapter 7, "Data Integrity". 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.

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 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 “User Tablespace Settings and Quotas” on page 17–6.

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 is operating system dependent.

OSDoc

Additional Information: See your Oracle operating system–specific documentation for more 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.

Though Oracle accepts the keywords ASC, DESC, COMPRESS, and NOCOMPRESS in the CREATE INDEX command, 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; for more information you can refer to computer science texts dealing with data structures. Figure 5 – 7 illustrates the structure of a B*–tree index.

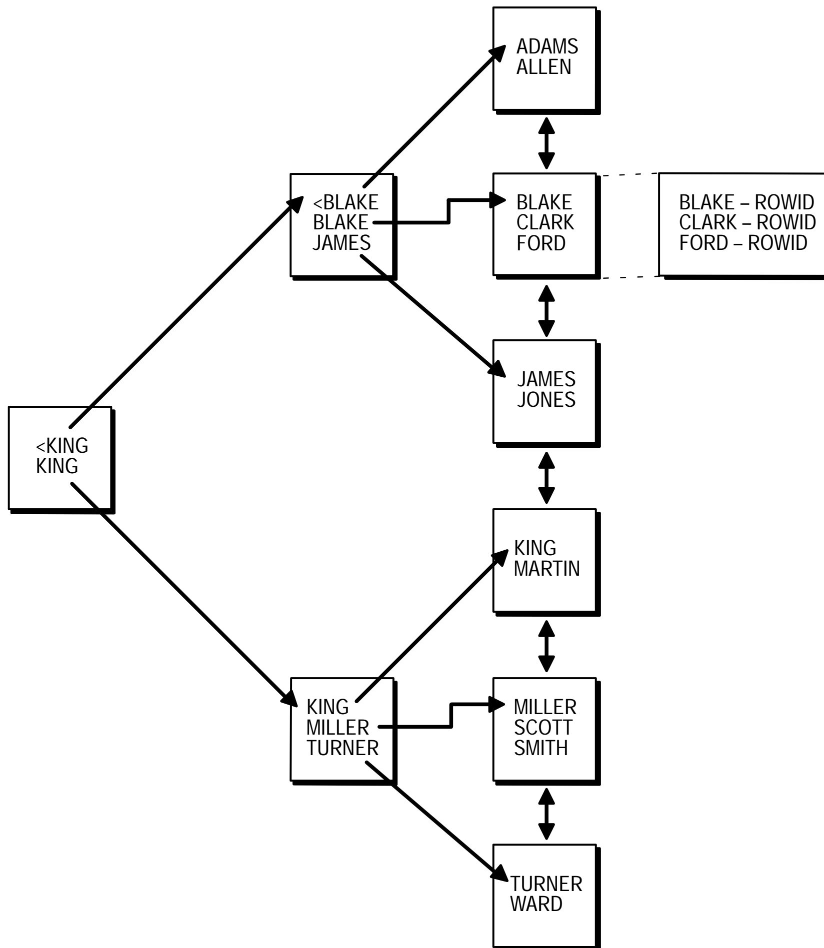


Figure 5 – 7 Internal Structure of a B*–Tree Index

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 non–unique index, the ROWID is included in the key in sorted order, so non–unique 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 and not violate a unique index.

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.

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 (see Figure 5 – 8), Oracle physically stores all rows for each department from both the EMP and DEPT tables in the same data blocks.

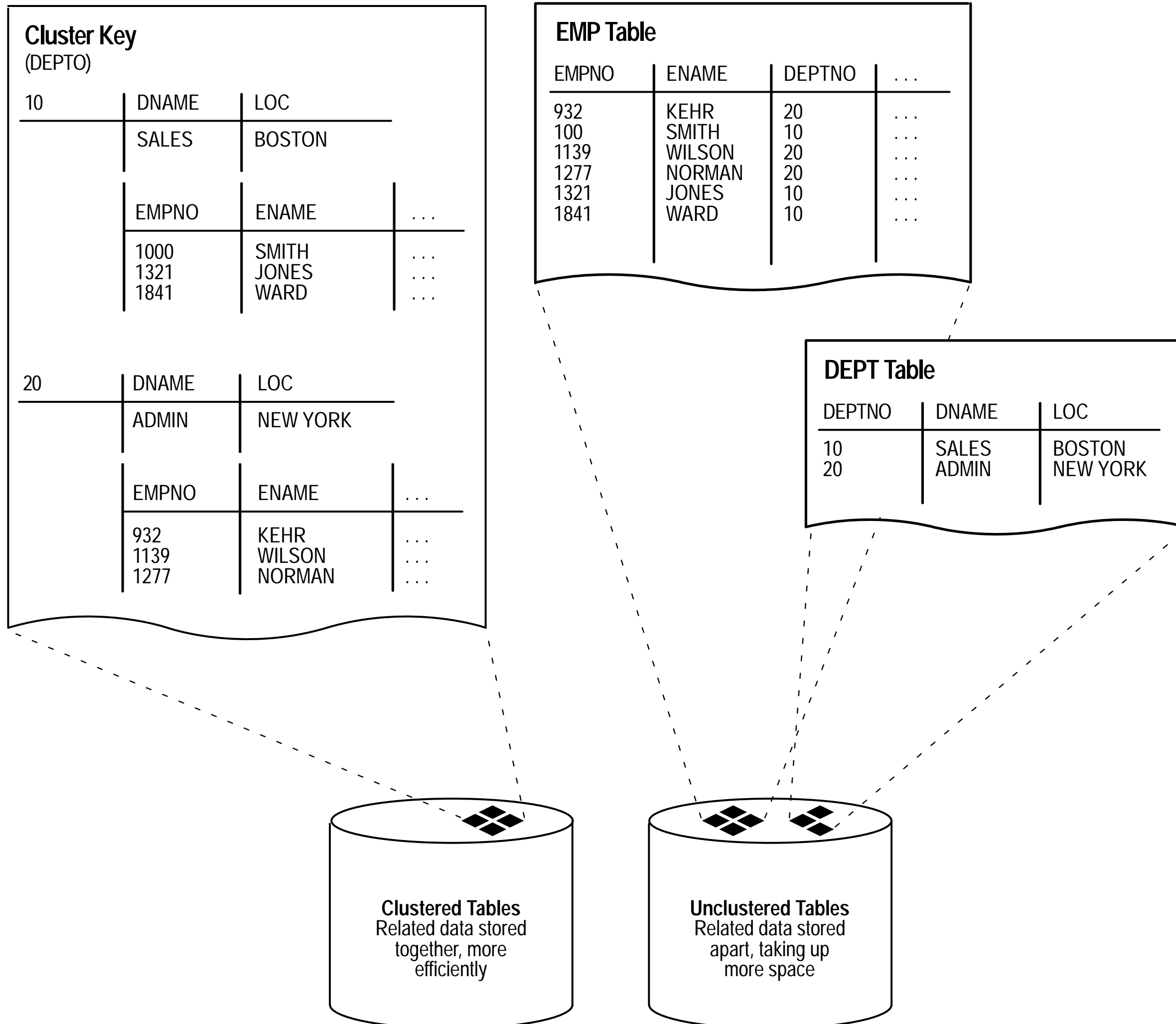


Figure 5 – 8 Clustered Table Data

Because clusters store related rows of different tables together in the same data blocks, properly used clusters offer two primary benefits:

- Disk I/O is reduced and 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 might be required to store related table and index data in a cluster than is necessary in non–clustered table format. For example, 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 as 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 non–clustered.

To identify data that would be better stored in clustered form than non–clustered, look for tables that are related via 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; 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 non–clustered table.

For more information about the performance implications of using clusters, see *Oracle7 Server Tuning*.

Format of Clustered Data Blocks

In general, clustered data blocks have an identical format to non–clustered 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 command. 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, 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, 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, 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 re–create 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, you 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 non–clustered 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 (but often more): one or more I/Os to find or store the key value in the index, and 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 5 – 9 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.

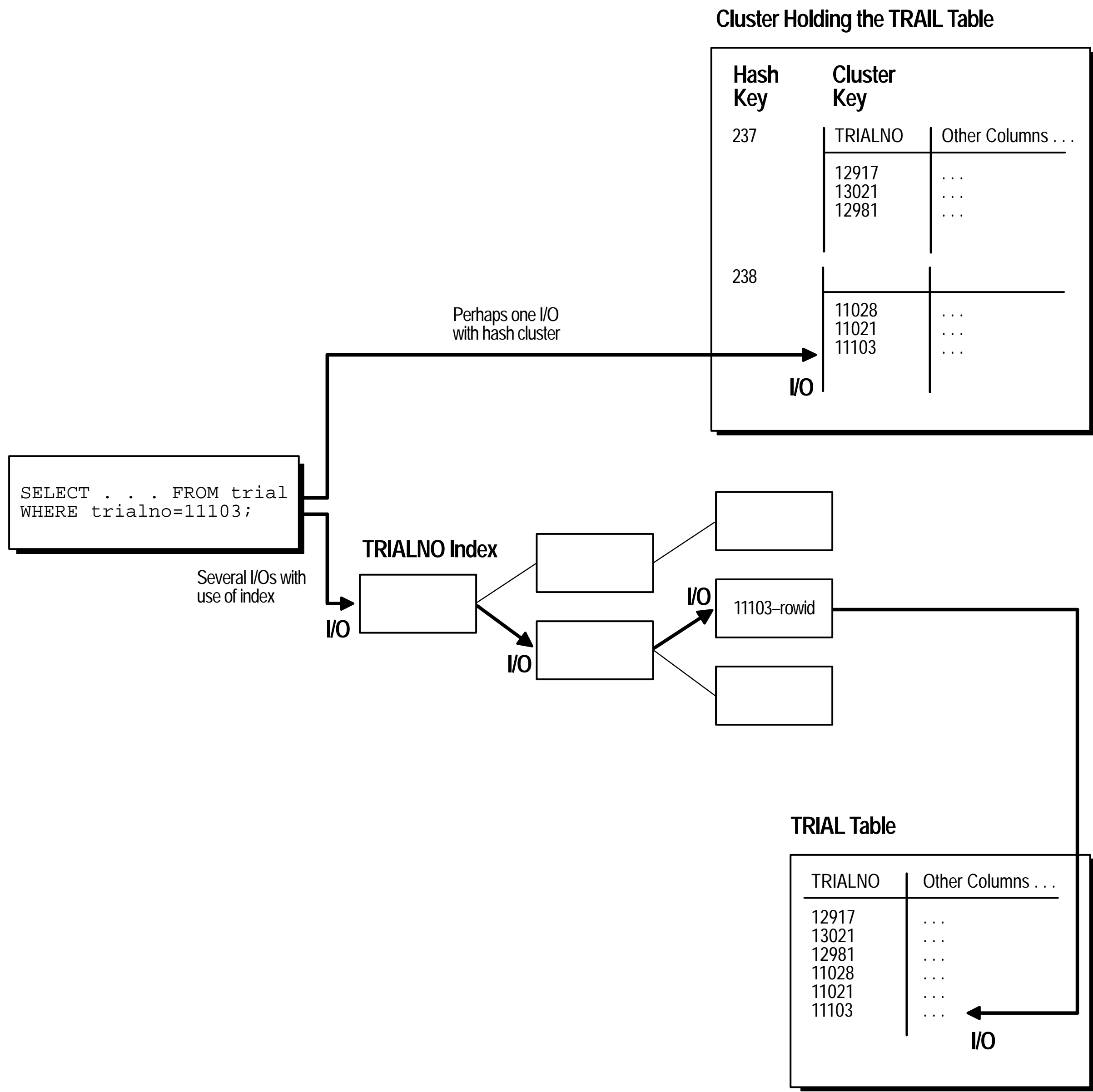


Figure 5 – 9 Hashing vs. Indexing: Data Storage and Information Retrieval

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 command.

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 command. 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, 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, 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.

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 be comprised 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, 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 command 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, 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, you should consider creating your own hash function that more efficiently distributes cluster rows 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, 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 command:

```
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 is also dependent 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 might or might not 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, an overflow block must be allocated to hold the new row. The likelihood of this happening is largely dependent on the average size of each hash key value and corresponding data, specified when the hash cluster is created, as illustrated in Figure 5 – 10.

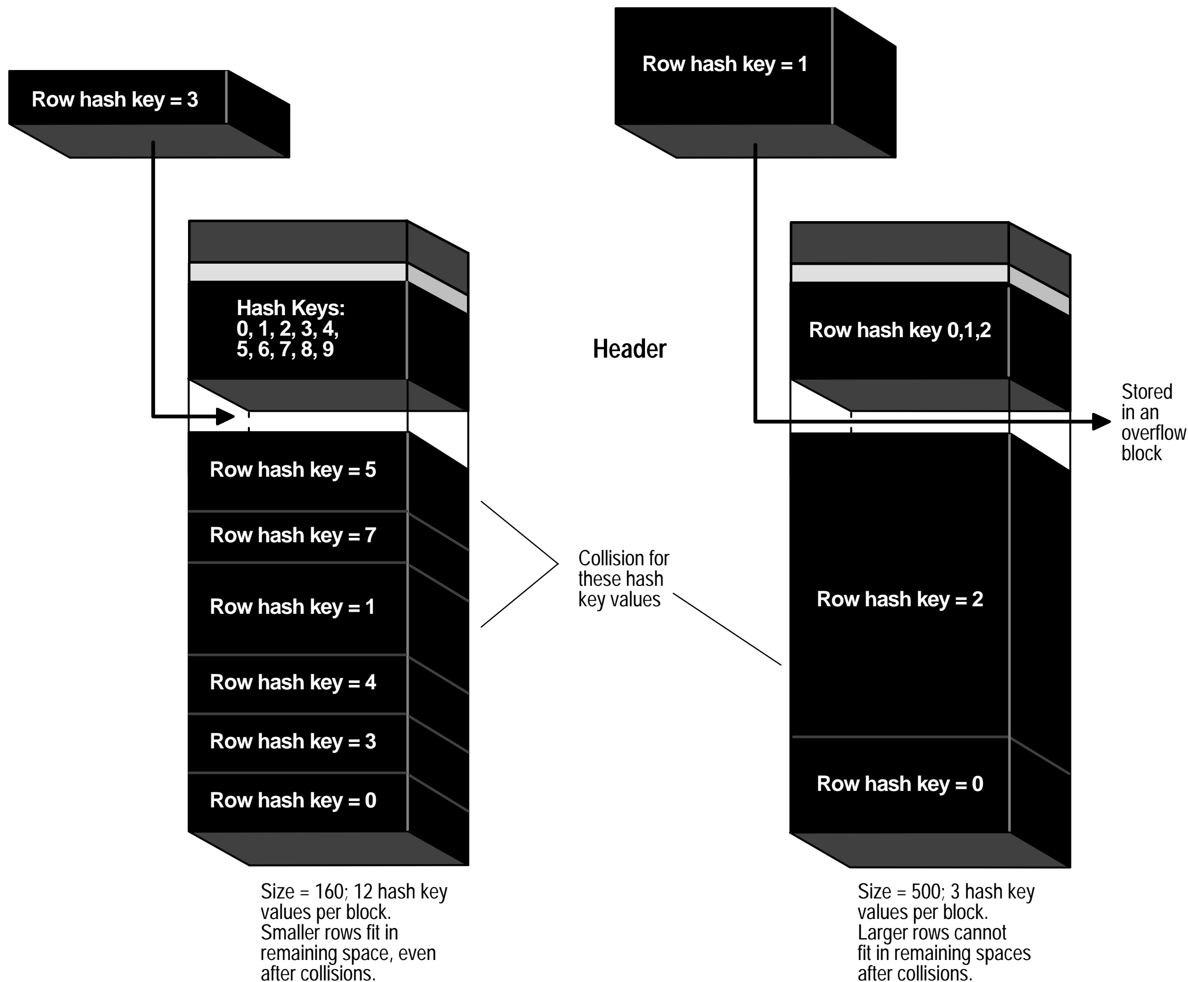


Figure 5 – 10 Collisions and Overflow Blocks in a Hash Cluster

If the average size is small and each row has a unique hash key value, 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, only a few hash key values can be assigned per data block. In this case, it is likely that the large row will not be able to fit in the root block allocated for the hash key value and an overflow block is allocated.

CHAPTER

6

Datatypes

I am the voice of today, the herald of tomorrow.
...I am the leaden army that conquers the world — I am TYPE.

Frederic William Goudy: *The Type Speaks*

This chapter discusses the Oracle datatypes, their properties, and how they map to non–Oracle datatypes. It includes:

- Oracle Datatypes
- ANSI, DB2, and SQL/DS Datatypes
- Data Conversions

Oracle Datatypes

The following sections describe the Oracle datatypes that you can use in column definitions:

- CHAR
- VARCHAR2
- VARCHAR
- NUMBER
- DATE
- LONG
- RAW
- LONG RAW
- ROWID
- MLSLABEL

Character Datatypes

The CHAR and VARCHAR2 datatypes store alphanumeric data. Character data is stored 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 multi–byte encoding schemes.

CHAR Datatype

The CHAR datatype stores **fixed**–length character strings. When you create a table with a CHAR column, you must specify a column length (in bytes, not characters) between 1 and 255 for the CHAR column (default is 1). Oracle then guarantees the following:

- 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, the value is blank–padded to the fixed length.
- If you give a longer value with trailing blanks, 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 the **blank–padded comparison semantics**. See *Oracle7 Server SQL Reference* for more information on comparison semantics.

VARCHAR2 Datatype

The VARCHAR2 datatype stores variable–length character strings. When you create a table with a VARCHAR2 column, you specify a maximum column length (in bytes, not characters) between 1 and 2000 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 and 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 only stores the 10 characters (10 bytes), not 50.

Oracle compares VARCHAR2 values using the **non–padded comparison semantics**. See *Oracle7 Server SQL Reference* for more information on comparison semantics.

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. However, in a future version of Oracle, the VARCHAR datatype might store variable–length character strings compared with different comparison semantics. Therefore, use the VARCHAR2 datatype to store variable–length character strings.

How to Choose the Correct Character Datatype

When deciding which datatype to use for a column that will store alphanumeric data in a table, consider the following points of distinction:

- **Comparison Semantics**
  Use the CHAR datatype when you require ANSI compatibility in comparison semantics, that is, when trailing blanks are not important in string comparisons. Use the VARCHAR2 when trailing blanks are important in string comparisons.
- **Space Usage**
  To store data more efficiently, use the VARCHAR2 datatype. The CHAR datatype blank–pads and stores trailing blanks up to a fixed column length for all column values, while the VARCHAR2 datatype does not blank–pad or store trailing blanks for column values.
- **Future Compatibility**
  The CHAR and VARCHAR2 datatypes are and will always be fully supported. At this time, the VARCHAR datatype automatically corresponds to the VARCHAR2 datatype and is reserved for future use.

Column Lengths for Character Datatypes and NLS Character Sets

The National Language Support (NLS) feature of Oracle allows the use of various character sets for the character datatypes. You should consider the size of characters when you specify the column length for character datatypes. For example, some characters require one byte, some require two bytes, and so on. You must consider this issue when estimating space for tables with columns that contain character data. See *Oracle7 Server Reference* and *Oracle7 Server Utilities* for more information about the NLS feature of Oracle.

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 x 10^–130 to 9.99..9 x 10^125 (with up to 38 significant digits)
- negative numbers from –1 x 10^–130 to 9.99..99 x 10^125 (with up to 38 significant digits)
- zero
- positive and negative infinity (generated only in import from a Version 5 database)

For numeric columns you can specify the column as follows:

```
column_name NUMBER
```

Optionally, you can also specify a *precision* (total number of digits) and *scale* (number of digits to right of 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 6 – 1 shows examples of how data would be stored using different scale factors.

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

Table 6 – 1 How Scale Factors Affect Numeric Data Storage

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 6 – 1.

For input and output of numbers, the standard Oracle default decimal character is a period, as in the number "1234.56". (The decimal is the character that separates the integer and decimal parts of a number.) You can change the default decimal character with the 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. (However, there are only 38 digits of precision.) Oracle does not store leading and trailing zeros. For example, the number 412 is stored in a format similar to 4.12 x 10^2, with one byte used to store the exponent (2) and two bytes used to store the three significant digits of the mantissa (4, 1, 2).

Taking this into account, the column data size for a particular numeric data value NUMBER (*p*), where *p* is the precision of a given value (scale has no effect), can be calculated using the following formula:

```
  1 byte                (exponent)
  FLOOR(p/2)+1 bytes    (mantissa)
+ 1 byte                (only for a negative number where
                        the number of significant digits is
                        less than 38)
_______________________
number of bytes of data
```

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, while 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 ranging from Jan 1, 4712 BC through Dec 31, 4712 AD. Unless you specifically specify BC, AD 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 indicates 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 12:00:00 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, the TO_DATE function must be used 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 from a common reference. (The reference is 01–01–4712 years B.C., so current dates are somewhere in the 2.4 million range.) A Julian date is nominally a non–integer, 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.

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).

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. Also see the next section for information about the LONG RAW datatype.

Uses of LONG Data

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.

Restrictions on LONG and LONG RAW Data

Although LONG (and LONG RAW; see below) columns have many uses, there are some restrictions on their use:

- Only one LONG column is allowed per table.
- LONG columns cannot be indexed.
- LONG columns cannot appear in integrity constraints.

- LONG columns cannot be used in WHERE, GROUP BY, ORDER BY, CONNECT BY clauses, or with the DISTINCT operator in SELECT statements.
- LONG columns cannot be referenced by SQL functions (such as SUBSTR or INSTR).
- LONG columns cannot be used in the SELECT list of a subquery or queries combined by set operators (UNION, UNION ALL, INTERSECT, or MINUS).
- LONG columns cannot be used in expressions.
- LONG columns cannot be referenced when creating a table with a query (CREATE TABLE . . . AS SELECT . . .) or when inserting into a table (or view) with a query (INSERT INTO . . . SELECT . . .).
- A variable or argument of a PL/SQL program unit cannot be declared using the LONG datatype.
- Variables in database triggers cannot be declared using the LONG or LONG RAW datatypes.
- References to :NEW and :OLD in database triggers cannot be used with LONG or LONG RAW columns.

RAW and LONG RAW Datatypes

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 is dependent on the use.

RAW is a variable–length datatype like the VARCHAR2 character datatype, except that SQL*Net (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, SQL*Net and Import/Export automatically convert CHAR, VARCHAR2, and LONG data between the database character set to the user session character set (set by the NLS_LANGUAGE parameter of the ALTER SESSION command), 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.

ROWIDs and the ROWID Datatype

Every row in a non–clustered table of an Oracle database is assigned a unique *ROWID* that corresponds to the physical address of a row's row piece (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.

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:

```
SELECT ROWID, ename FROM emp;
```

ROWIDs use a binary representation of the physical address for each row selected. When queried using SQL*Plus or Server Manager, the binary representation is converted to a VARCHAR2/hexadecimal representation. The above query might return the following row information:

```
ROWID              ENAME
------------------ ----------
00000DD5.0000.0001 SMITH
00000DD5.0001.0001 ALLEN
00000DD5.0002.0001 WARD
```

As shown above, a 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.

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.

You cannot set the value of the pseudocolumn ROWID in INSERT or UPDATE statements. Oracle uses the ROWIDs in the pseudocolumn ROWID internally for various operations as described in the following section. Though you can reference ROWIDs in the pseudocolumn ROWID like other table columns (used in SELECT lists and WHERE clauses), ROWIDs are not stored in the database, nor are they database data.

ROWIDs and 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 binary 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 non–standard translation to hexadecimal format of length up to 256 bytes. Refer to the relevant manual for OCIs or Precompilers for further details on handling ROWIDs with non–Oracle systems.

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 uses:

- 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.

Examples of Using ROWIDs

Using some group functions with ROWID, you can see how data is internally stored in an Oracle database.

Use the function SUBSTR to break the data in ROWID into its three components (file, block, and row). For example:

```
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 tells how many datafiles contain rows of a given table:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID,15,4))) "FILES" FROM tablename;

   FILES
--------
       2
```

For more information on how to use ROWIDs, refer to the *Oracle7 Server SQL Reference*, the *PL/SQL User's Guide and Reference*, *Oracle7 Server Tuning*, and other books that document Oracle tools and utilities.

The MLSLABEL Datatype

Trusted Oracle provides one additional datatype: the MLSLABEL datatype. You can declare columns of the MLSLABEL datatype in standard Oracle, as well as in Trusted Oracle, for compatibility with Trusted Oracle applications.

The MLSLABEL datatype is used to store the binary format of an operating system label. The maximum width of a column declared as MLSLABEL is 255 bytes.

MLSLABEL data is stored as a variable–length tag (two to five bytes, in which the first byte indicates length) that maps to a binary label in the data dictionary. The reason that MLSLABEL data is stored as a representative tag instead of the binary label itself is that binary operating system labels can be very long. Given that a label is stored with every row in the database (in the ROWLABEL column), storing many large labels can consume a lot of space. Storing a representative tag instead of a label is more space efficient.

Any labels that are valid on your operating system can be inserted into an MLSLABEL column. When you insert a label into an MLSLABEL

column, Trusted Oracle implicitly converts the data into the binary format of the label.

The following sections further describe this datatype and the ROWLABEL pseudocolumn. If you are using Trusted Oracle, also see the *Trusted Oracle7 Server Administrator's Guide* for more information.

The ALL_LABELS Data Dictionary View

The ALL_LABELS data dictionary view lists all of the labels ever stored in the database, including the values of DBHIGH and DBLOW. Any label ever stored in an MLSLABEL column (including the ROWLABEL column) is automatically added to this view.

Note that this view does not necessarily contain all labels that are valid in the database, since any valid operating system label, in any valid format, is a valid label within Trusted Oracle. Also note that this view may contain labels that are invalid within the database (if those labels were once used in the database, but are no longer valid).

The ROWLABEL Column

Oracle automatically appends the ROWLABEL column to each Trusted Oracle table at table creation. This column contains a label of the MLSLABEL datatype for every row in the table.

In OS MAC mode, given that a table can contain rows at one label only, the values in this column are always uniform within a table (and within a single database).

In DBMS MAC mode, the values in this column can range within a single table from DBHIGH to DBLOW (within any constraints defined for that table).

Summary of Oracle Datatype Information

For quick reference, Table 6 – 2 summarizes the important information about each Oracle datatype.

Data Type	Description	Column Length (bytes)
CHAR (*size*)	Fixed–length character data of length *size.*	Fixed for every row in the table (with trailing blanks); maximum size is 255 bytes per row, default size is one byte per row. Consider the character set that is used before setting *size.* (Are you using a one–byte or multi–byte character set?)
VARCHAR2 (*size*)	Variable–length character data. A maximum *size* must be specified.	Variable for each row, up to 2000 bytes per row. Consider the character set that is used before setting *size.* (Are you using a one–byte or multi–byte character set?)
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 B.C. to December 31, 4712 A. D. Default format: DD–MON–YY.	Fixed at seven bytes for each row in the table.
LONG	Variable–length character data.	Variable for each row in the table, up to 2^31 – 1 bytes, or two gigabytes, per row.
RAW (*size*)	Variable–length raw binary data. A maximum *size* must be specified.	Variable for each row in the table, up to 255 bytes per row.
LONG RAW	Variable–length raw binary data.	Variable for each row in the table, up to 2^31 – 1 bytes, or two gigabytes, per row.
ROWID	Binary data representing row addresses.	Fixed at six bytes for each row in the table.
MLSLABEL	Variable–length binary data representing operating system labels.	Variable for each row in the table, ranging from two to five bytes per row.

Table 6 – 2 Summary of Oracle Datatype Information

ANSI, DB2, and SQL/DS Datatypes

In addition to Oracle datatypes, columns of tables in an Oracle database can be defined using ANSI, DB2, and SQL/DS datatypes. However, Oracle internally converts such datatypes to Oracle datatypes.

ANSI SQL Datatype	Oracle Datatype
CHARACTER (n), CHAR(n)	CHAR(n)
NUMERIC (p, s), DECIMAL (p, s) DEC (p, a)	NUMBER (p, s)
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT (p), REAL, DOUBLE PRECISION	NUMBER
CHARACTER VARYING(n), CHAR VARYING(n)	VARCHAR(n)

Table 6 – 3 ANSI 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)	NUMBER
DATE	DATE

Table 6 – 4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

The IBM products SQL/DS and DB2 datatypes TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC have no corresponding Oracle datatype, and cannot be used. The TIME and TIMESTAMP datatypes are subcomponents of the Oracle datatype DATE.

The ANSI datatypes NUMERIC, DECIMAL, and DEC can specify only fixed–point numbers. For these datatypes, *s* defaults to 0.

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()
- TO_LABEL()
- CHARTOROWID()
- ROWIDTOCHAR()
- HEXTOCHAR()
- CHARTOHEX()

Implicit datatype conversions work according to the rules explained in the following two sections.

> **Note:** If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for additional information involving data conversions and the MLSLABEL and RAW MLSLABEL datatypes.

Rule 1: Assignments

For assignments, Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- NUMBER to VARCHAR2
- VARCHAR2 or CHAR to DATE
- DATE to VARCHAR2
- VARCHAR2 or CHAR to ROWID
- ROWID to VARCHAR2
- VARCHAR2 or CHAR to LABEL
- LABEL to VARCHAR2
- VARCHAR2 or CHAR to HEX
- HEX to VARCHAR2

The assignment succeeds if Oracle can convert the datatype of the value used in the assignment to that of the assignment's target.

Note: For the examples in the following list, assume a package with a public variable and a table declared as in the following statements:

```
var1    CHAR(5);
CREATE TABLE table1 (col1 NUMBER);
```

- variable := expression

  The datatype of *expression* must be either be the same as or convertible to the datatype of *variable*. For example, Oracle automatically converts the data provided in the following assignment within the body of a stored procedure:

  ```
  VAR1 := 0
  ```

- INSERT INTO *table* VALUES (*expression1*, *expression2*, ...)

  The datatypes of *expression1*, *expression2*, and so on, must either be the same as or convertible to the datatypes of the corresponding columns in *table*. For example, Oracle automatically converts the data provided in the following INSERT statement for TABLE1 (see table definition above):

  ```
  INSERT INTO table1 VALUES ('19');
  ```

- UPDATE *table* SET *column* = *expression*

  The datatype of *expression* must either be the same as or convertible to the datatype of *column*. For example, Oracle automatically converts the data provided in the following UPDATE statement issued against TABLE1:

  ```
  UPDATE table1 SET col1 = '30';
  ```

- SELECT *column* INTO *variable* FROM *table*

  The datatype of *column* must either be the same as or convertible to the datatype of *variable*. For example, Oracle automatically converts data selected from the table before assigning it to the variable in the following statement:

  ```
  SELECT col1 INTO var1 FROM table1 WHERE col1 = 30;
  ```

Rule 2: Expression Evaluation

For expression evaluation, Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- VARCHAR2 or CHAR to DATE

Some common types of expressions follow:

- Simple expressions, such as the following:

  comm + ’500’

- Boolean expressions, such as the following:

  bonus > sal / ’10’

- Function and procedure calls, such as the following:

  MOD (counter, ’2’)

- WHERE clause conditions, such as the following:

  WHERE hiredate = ’01–JAN–91’

In general, Oracle uses the rule for expression evaluation when a datatype conversion is needed in places not covered by the rule for assignment conversions.

In assignments of the form

```
variable := expression
```

Oracle first evaluates *expression* using the conversions covered by Rule 2; *expression* can be simple or complex. If it succeeds, *expression* results in a single value and datatype. Then, Oracle tries to assign this value to the assignment’s target using Rule 1.

CHAR to NUMBER conversions only succeed if the character string represents a valid number. CHAR to DATE conversions only succeed if the character string has the default format ’DD–MON–YY’.

CHAPTER

7

Data Integrity

Does one's integrity ever lie in what he is not able to do?

Flannery O'Connor: *Wise Blood*

This chapter explains how to enforce the business rules associated with your database and prevent the entry of invalid information into tables using integrity constraints. The chapter includes:

- Definition of Data Integrity
- An Introduction to Integrity Constraints
- Types of Integrity Constraints
- The Mechanisms of Constraint Checking

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for more information about integrity constraints in that environment.

Definition of 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 7 – 1.

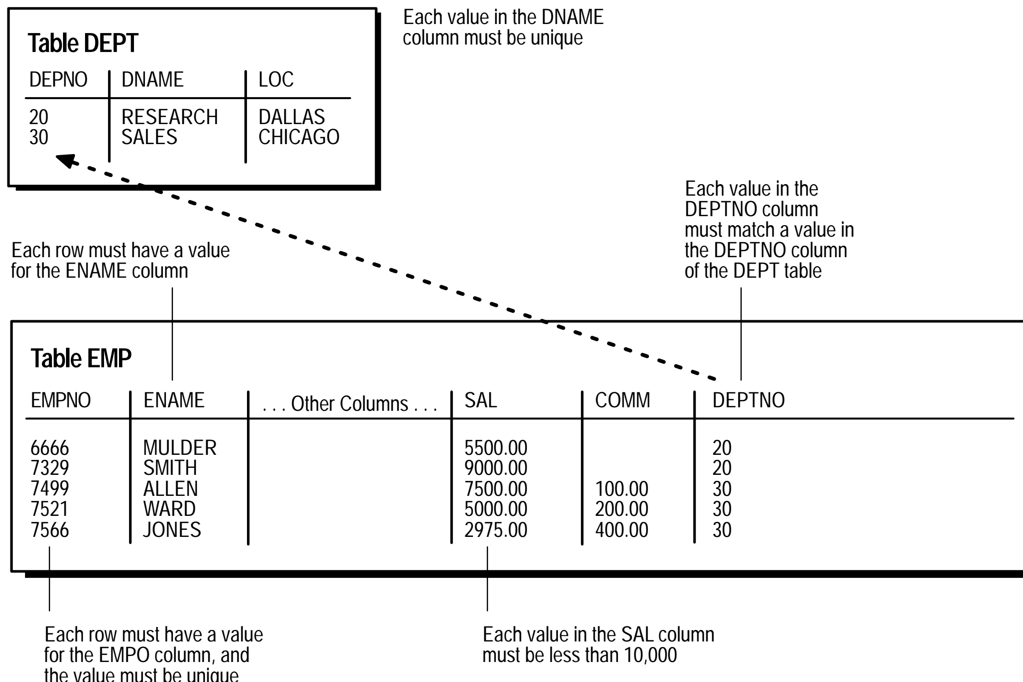


Figure 7 – 1 Examples of Data Integrity

Note that certain columns of each table have specific rules that constrain the data contained within them.

Types of Data Integrity

The following types of rules are applied to tables and enable you to enforce different types of data integrity.

Nulls

A rule defined on a single column that allows or disallows inserts or updates of rows containing a null for the column.

Unique Column Values

A rule defined on a column (or set of columns) that allows only the insert or update of a row containing a unique value for the column (or set of columns).

Primary Key Values

A rule defined on a column (or set of columns) so that each row in the table can be uniquely identified by the values in the column (or set of columns).

Referential Integrity

A rule defined on a column (or set of columns) in one table that allows the insert or update of a row only if the value for the column or set of columns (the dependent value) matches a value in a column of 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 include:

Restrict	A referential integrity rule that 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.

Complex Integrity Checking

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 allows you to define and enforce each type of the data integrity rules defined in the previous section. Most of these rules are easily defined using integrity constraints.

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 integrity constraints for the rules associated with nulls in a column
- UNIQUE key integrity constraints for the rule associated with unique column values
- PRIMARY KEY integrity constraints for the rule associated with primary identification values

- FOREIGN KEY integrity 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 RESTRICT
  - delete CASCADE
- CHECK integrity constraints for complex integrity rules

Other referential integrity actions not included on this list can be defined using database triggers (see the following section).

> **Note:** You cannot enforce referential integrity using declarative integrity constraints if child and parent tables are on different nodes of a distributed database. However, you can enforce referential integrity in a distributed database using database triggers (see next section).

Database Triggers

Oracle also allows you to enforce integrity rules with a non–declarative approach using database triggers (stored database procedures automatically invoked on insert, update, or delete operations). While database triggers allow you to define and enforce any type of integrity rule, it is strongly recommended that you use database triggers only in the following situations:

- to enforce referential integrity when a required referential integrity rule cannot be enforced using the integrity constraints listed above: update CASCADE, update and delete SET NULL, update and delete SET DEFAULT
- 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

For more information and examples of database triggers used to enforce data integrity, see Chapter 15, “Database Triggers”.

An 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 that are associated with the information in a database. If any of the results of a DML statement execution violate an integrity constraint, 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, Oracle rolls back the statement and returns an informative error.

The integrity constraints implemented in Oracle fully comply with the standards set forth by ANSI X3.135–1989 and ISO 9075–1989.

Advantages of Integrity Constraints

Integrity constraints are not the only way to enforce data integrity rules on the data of your database. You can also

- enforce business rules in the code of a database application
- use stored procedures to completely control access to data
- enforce business rules using triggered stored database procedures (see Chapter 15, "Database Triggers")

The following section describes some of the advantages that integrity constraints have over these other alternatives.

Declarative Ease

Because you define integrity constraints using SQL commands, when you define or alter a table, no programming is required. Therefore, they are easy to write, eliminate programming errors, and 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, unlike the stored procedure solution to data integrity by controlled data access, integrity constraints do not eliminate the flexibility of ad hoc data access.

Centralized Rules

Integrity constraints are defined for tables (not an application) and stored in the data dictionary. Therefore, the data entered by any application must adhere to the same integrity constraints associated with a 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, and although database triggers can provide this benefit, 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, the administrator need only change that integrity constraint and all applications automatically adhere to the modified constraint. Alternatively, if a business rule is enforced by the code of each database application, developers must modify all application source code and recompile, debug, and test the modified applications.

Immediate User Feedback

Because 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

Because the semantics of integrity constraint declarations are clearly defined, 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 done when necessary.)

Flexibility for Data Loads and Identification of Integrity Violations

Integrity constraints can be temporarily disabled 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

The integrity constraints that you can use to impose restrictions on the input of column values can be of the following types:

- NOT NULL constraints
- UNIQUE key constraints
- PRIMARY KEY constraints
- FOREIGN KEY (referential) constraints
- CHECK constraints

The following sections explain each type of integrity constraint in detail. The information in each section includes the following:

- the rule enforced by the constraint
- an example of the constraint
- recommendations for appropriate use of the constraint
- other important information about the constraint

NOT NULL Integrity Constraints

By default, all columns in a table allow nulls (the absence of a value). A NOT NULL constraint requires that no nulls be allowed in a column of a table. 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 7 – 2 illustrates a NOT NULL integrity constraint.

Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9,000.00		20
7499	ALLEN	VP_SALES	7329	20–FEB–90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90	2,975.00	400.00	30

NOT NULL Constraint
(no row may contain a null value for this column)

Absence of NOT NULL Constraint
(any row can contain null for this column)

Figure 7 – 2 NOT NULL Integrity Constraints

UNIQUE Key Integrity Constraints

A UNIQUE key integrity constraint requires that no two rows of a table have duplicate values in a specified column or set of columns.

For example, consider the DEPT table in Figure 7 – 3. A UNIQUE key constraint is defined on the DNAME column to disallow rows with duplicate department names.

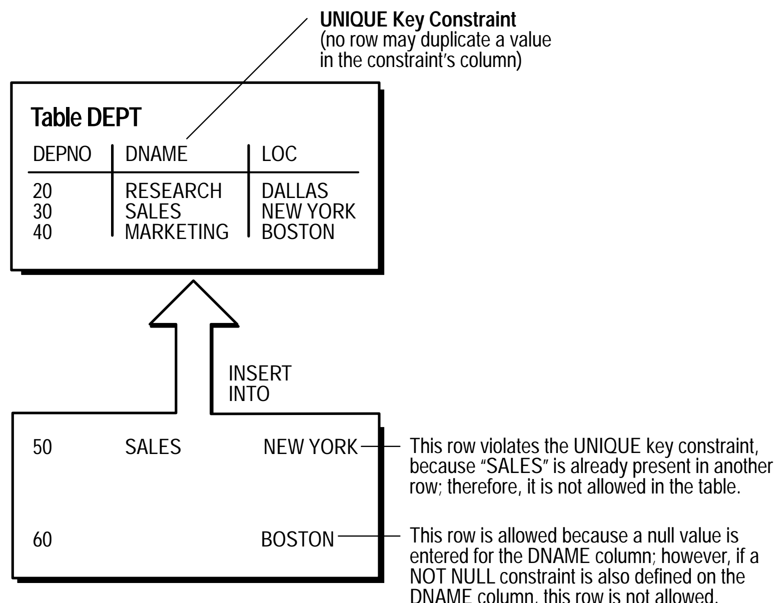


Figure 7 – 3 A UNIQUE Key Constraint

Unique Keys

The column (or set of columns) included in the definition of the UNIQUE key constraint is called the *unique key*. The term "unique key" is often incorrectly used as a synonym for the terms "UNIQUE key constraint" or "UNIQUE index"; however, note that the term "key" refers only to the list of columns used in the definition of the integrity constraint.

If the UNIQUE key constraint is comprised of more than one column, that group of columns is said to be a composite unique key. For example, in Figure 7 – 4, the CUSTOMER table has a UNIQUE key constraint defined on the composite unique key: the AREA and PHONE columns.

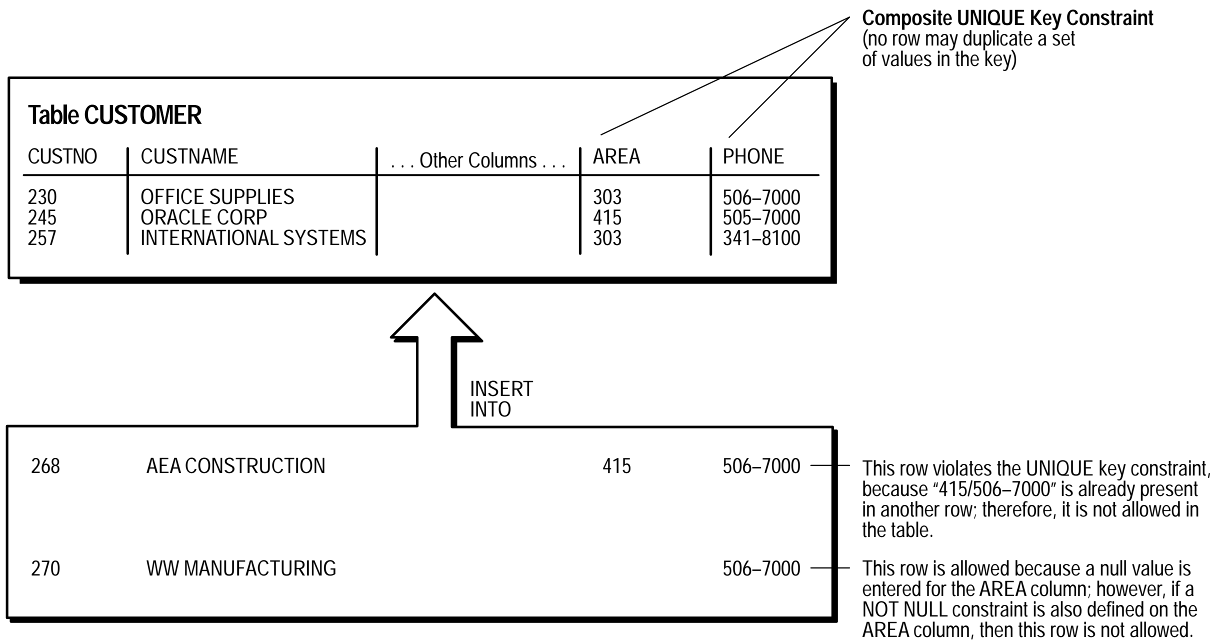


Figure 7 – 4 A Composite UNIQUE Key Constraint

This UNIQUE key constraint allows you to enter an area code and phone number any number of times, but the combination of a given area code and given phone number cannot be duplicated in the table. This eliminates unintentional duplication of a phone number.

UNIQUE Key Constraints and Indexes

Oracle enforces unique integrity constraints with indexes. In Figure 7 – 4, Oracle enforces the UNIQUE key constraint by implicitly creating a unique index on the composite unique key. Because Oracle enforces UNIQUE key constraints using indexes, composite UNIQUE key constraints are limited to the same limitations imposed on composite indexes: up to 16 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.

Combining UNIQUE Key and NOT NULL Integrity Constraints

Notice in the examples of the previous section that 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. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE key constraint.

It is common to define unique keys on columns with NOT NULL integrity constraints. This combination forces the user to input values in the unique key; this combination of data integrity rules eliminates the possibility that any new row's data will ever risk conflicting 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 column (or set of columns) included in the definition of a table's PRIMARY KEY integrity constraint is 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 7 – 5 illustrates a PRIMARY KEY constraint in the DEPT table and examples of rows that the constraint prevents from entering the table.

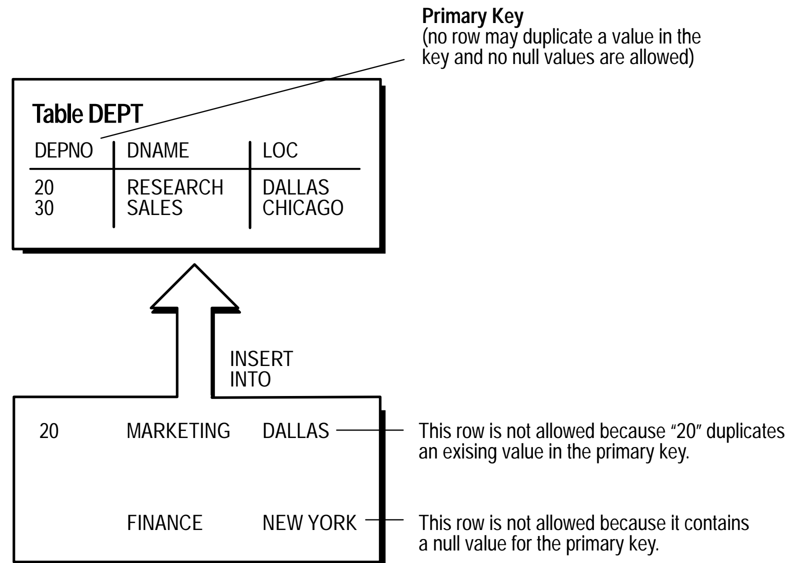


Figure 7 – 5 A Primary Key Constraint

PRIMARY KEY Constraints and Indexes

Oracle enforces all PRIMARY KEY constraints using indexes. In the previous example, the primary key constraint created for the DEPTNO column is enforced by

- the implicit creation of a unique index on that column
- the implicit creation of a NOT NULL constraint for that column

Because Oracle enforces primary key constraints using indexes, composite primary key constraints are limited to 16 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.

Referential Integrity and FOREIGN KEY (Referential) Integrity Constraints

Because tables of a relational database can be related by common columns, the rules that govern the relationship of the columns must be maintained. Referential integrity rules guarantee that these relationships are preserved.

There are several terms 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* (see the following).

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	A dependent or child table is 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	A referenced or parent table is 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 7 – 6 illustrates the above terms.

Figure 7 – 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 (the DEPTNO column). Therefore, no erroneous department numbers can exist in the DEPTNO column of the EMP table.

Foreign keys can be comprised of multiple columns. However, a composite foreign key must reference a composite primary or unique key with the same number of columns and datatypes. Because composite primary and unique keys are limited to 16 columns, a composite foreign key is also limited to 16 columns.

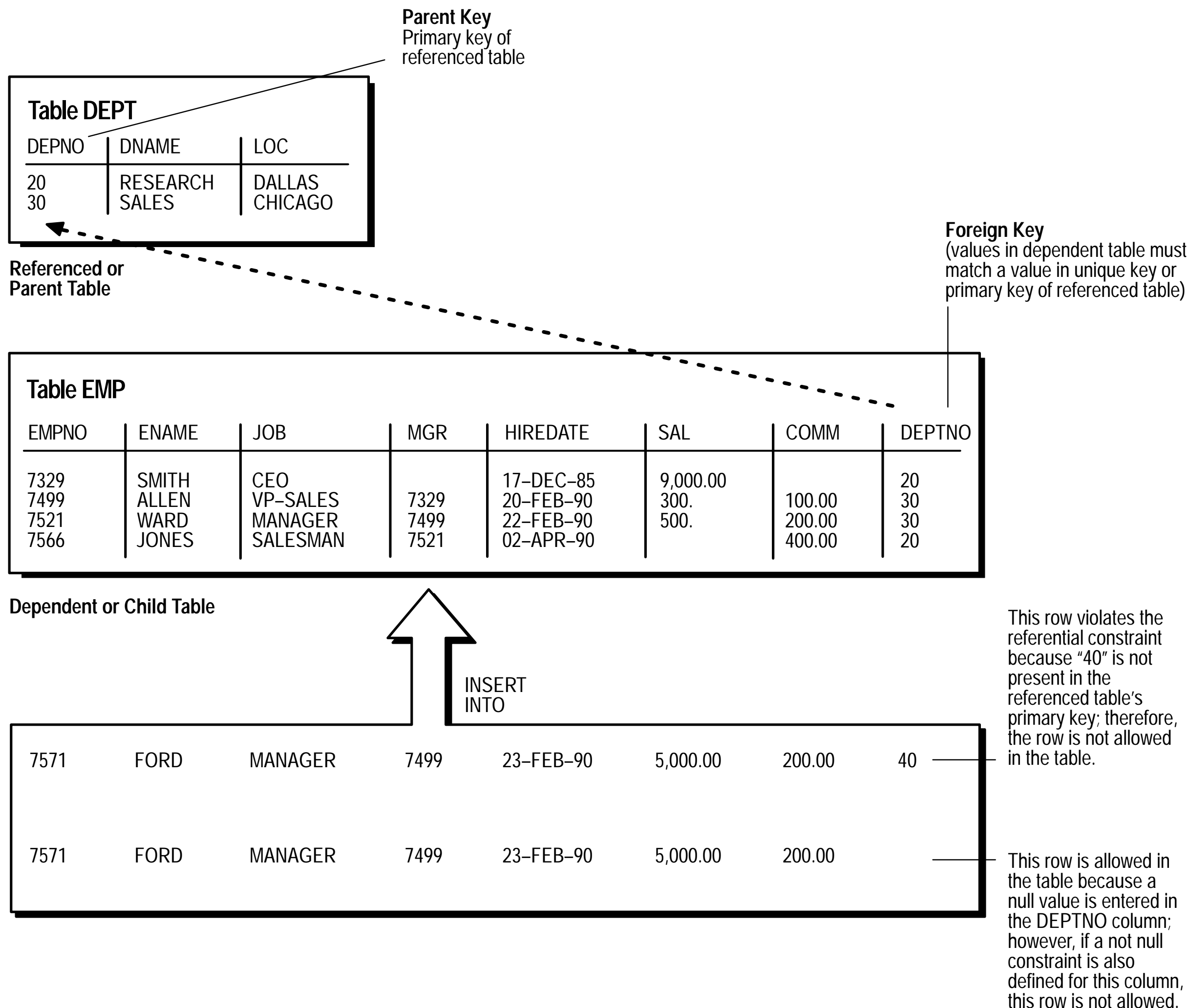


Figure 7 – 6 Referential Integrity Constraints

Self–Referential Integrity Constraints

Another type of referential integrity constraint, shown in Figure 7 – 7, is called a self–referential integrity constraint. This type of foreign key references a parent key of the same table. In the example below, you define the referential integrity constraint so 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 (that is, every manager must also be an employee). This integrity constraint eliminates the possibility of erroneous employee numbers in the MGR column.

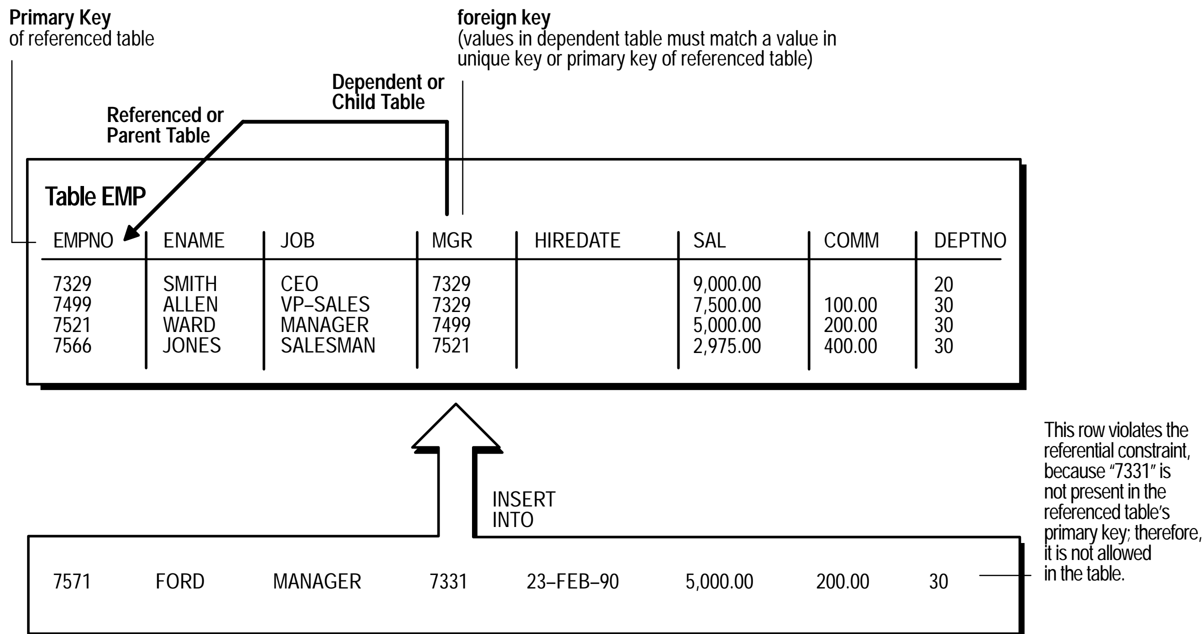


Figure 7 – 7 Single Table Referential Constraints

Nulls and Foreign Keys

The relational model permits foreign keys to be a value of the referenced primary or unique key, or null. There are several possible interpretations of this basic rule of the relational model 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 (for example, 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 also specify particular actions that are 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 include UPDATE and DELETE RESTRICT, and DELETE CASCADE.

> **Note:** Other referential actions not supported by FOREIGN KEY integrity constraints of Oracle can be enforced using database triggers. See Chapter 15, "Database Triggers," for more information regarding database triggers.

Update and Delete Restrict The restrict action 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, the referenced primary key value cannot be deleted because of the dependent data.

Delete Cascade The delete cascade action specifies that when rows containing referenced key values are deleted, all rows in child tables with dependent foreign key values are also deleted. Therefore, the delete cascades. 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, the rows in the child table that reference the primary key value are also deleted from the child table.

DML Restrictions with Respect to Referential Actions Table 7 – 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.

DML Statement	Issued Against Parent Table	Issued Against Child Table
INSERT	Always OK if parent key value is unique.	OK only if the foreign key value exists in the parent key or is partially or all null.
UPDATE Restrict	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 Restrict	Allowed if no rows in the child table reference the parent key value.	Always OK.
DELETE Cascade	Always OK.	Always OK.

Table 7 – 1 DML Statements Allowed by Update and Delete Restrict

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 is issued so that the condition of the CHECK constraint evaluates to false, the statement is rolled back.

The Check Condition

CHECK constraints allow you to enforce very specific or sophisticated integrity rules with the specification of a check condition. The condition of a CHECK constraint has some limitations, including that the condition must be a Boolean expression evaluated using the values in the row being inserted or updated, and cannot contain subqueries, sequences, the SYSDATE, UID, USER, or USERENV SQL functions, 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. (For more information on NLS features, see *Oracle7 Server Reference*.)

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.

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 illustrated in a previous example (see Figure 7 – 7 on page 7–14.).
- 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 only addresses 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? The three possibilities include the following:

- 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.
- The same value can be entered in both the EMPNO and MGR columns.
- 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.

Each case reveals something about how and when Oracle performs constraint checking.

The first case is easy to understand; a null is given for the foreign key value. Because nulls are allowed in foreign keys, this row is inserted successfully into the table.

The second case is more interesting. This case reveals when Oracle effectively 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.

The third case reveals even more about the constraint checking mechanism. This scenario shows that constraint checking is effectively deferred until the complete execution of the statement; all rows are inserted first, then all rows are checked for constraint violations.

As another example of this third case, consider the same self–referential integrity constraint and 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 7 – 8.

EMPNO	MGR
210	
211	210
212	211

Figure 7 – 8 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 7 – 9 shows that Oracle performs the actions of the entire SQL statement before any constraints are checked.

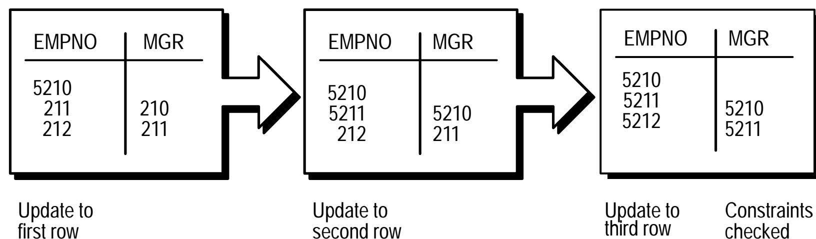


Figure 7 – 9 Constraint Checking

The examples in this section illustrated 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. However, the same mechanism is used for all types of constraints, including NOT NULL, UNIQUE key, PRIMARY KEY, all types of FOREIGN KEY, and CHECK constraints.

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.

CHAPTER

8

The Data Dictionary

LEXICOGRAPHER — A writer of dictionaries, a harmless drudge.

Samuel Johnson: *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:

- An Introduction to the Data Dictionary
- The Structure of the Data Dictionary
- SYS, the Owner of the Data Dictionary
- How the Data Dictionary Is Used
- The Dynamic Performance Tables

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. Even beginning users can benefit from understanding and using the data dictionary. However, no user should ever alter (update, delete, or insert) any rows or objects in the data dictionary; such activity can compromise data integrity.

An Introduction to the Data Dictionary

One of the most important parts of an Oracle database is its data dictionary. The *data dictionary* is a **read–only** set of tables that provides information about its associated database. For example, a data dictionary can provide the following information:

- the names of Oracle users
- privileges and roles each user has been granted
- names of schema objects (tables, views, snapshots, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- information about integrity constraints
- default values for columns
- how much space has been allocated for, and is currently used by, the objects in a database
- auditing information, such as who has accessed or updated various objects
- in Trusted Oracle, the labels of all objects and users (See the *Trusted Oracle7 Server Administrator's Guide*
- other general database information

The data dictionary is structured in tables and views, just like other database data. To access the data dictionary, you use SQL. Because the data dictionary is read–only, users can issue only queries (SELECT statements) against the tables and views of the data dictionary.

The Structure of the Data Dictionary

A database's data dictionary is comprised of

base tables	The foundation of the data dictionary is comprised of a set of base or underlying tables that store information about the associated database. Only Oracle should write 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 data dictionary contains user accessible views that summarize and conveniently display the information in the base tables of the dictionary. The views decode the information in the base tables into useful information, such as user or table names, and use 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 any object contained in the SYS schema and the security administrator should keep strict control of this central account.

> **Note:** 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 two primary uses:

- Oracle accesses the data dictionary every time that a DDL statement is issued.
- Any Oracle user can use the data dictionary as a read–only reference for information about the database.

How Oracle and Other Oracle Products Use 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 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 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.

Caching of the Data Dictionary for Fast Access

Because Oracle constantly accesses the data dictionary during database operation to validate user access and to verify the state of objects, much of the data dictionary information is cached in the SGA. 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 columns are not cached unless they are frequently accessed.

Other Programs and the Data Dictionary

Other Oracle products can create additional data dictionary tables or views of their own and reference existing views. 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. Never create new objects belonging to user SYS, except by running script provided by Oracle Corporation for creating data dictionary objects.

Deleting Data Dictionary Items

Because all changes to the data dictionary are performed by Oracle in response to DDL statements, **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 delete data from it because the rows are for information only and are not necessary for Oracle to run.

Public Synonyms for Data Dictionary Views

Public synonyms are created on many data dictionary views so they can be conveniently accessed by users. The security administrator can create additional public synonyms for objects used systemwide. However, other users should avoid naming their own objects with the same names as those used for public synonyms.

How Oracle Users Can Use the Data Dictionary

The views of the data dictionary serve as a reference for all database users. Access to the data dictionary views is via the SQL language. Certain views are accessible to all Oracle users, while others are intended for 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:

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 all users can access)

Table 8 – 1 Data Dictionary View Prefixes

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.

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 objects created by the user, grants made by the user, and so on
- display only rows pertinent to the user
- have identical columns 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 objects to which the user has access via public or explicit grants of privileges and roles, in addition to 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, as the DBA views should only be queried 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.

DUAL

The table named DUAL is a small table that Oracle and user–written programs can reference to guarantee a known result. This table has one column and one row.

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.*

Because dynamic performance tables are not true tables, 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.

SYS owns the dynamic performance tables and their names all begin with V_$. Views are created on these tables, and then synonyms are created for the views. The synonym names begin with V$.

PART

IV System Architecture

CHAPTER

9

Memory Structures and Processes

Yea, from the table of my memory
I'll wipe away all trivial fond records.

Shakespeare: *Hamlet*

This chapter discusses the memory structures and processes in an Oracle database system. It includes:

- An Oracle Instance
- Process Structure
- Oracle Memory Structures
- Variations in Oracle Configuration
- Examples of How Oracle Works
- The Program Interface

An Oracle Instance

Regardless of the type of computer executing Oracle and the particular memory and process options being used, every running Oracle database is associated with an Oracle instance. Every time a database is started on a database server, Oracle allocates a memory area called the System Global Area (SGA) and starts one or more Oracle processes. The combination of the SGA and the Oracle processes is called an Oracle database instance. The memory and processes of an instance work to manage the database's data efficiently and serve the one or multiple users of the associated database. Figure 9 – 1 shows an Oracle instance.

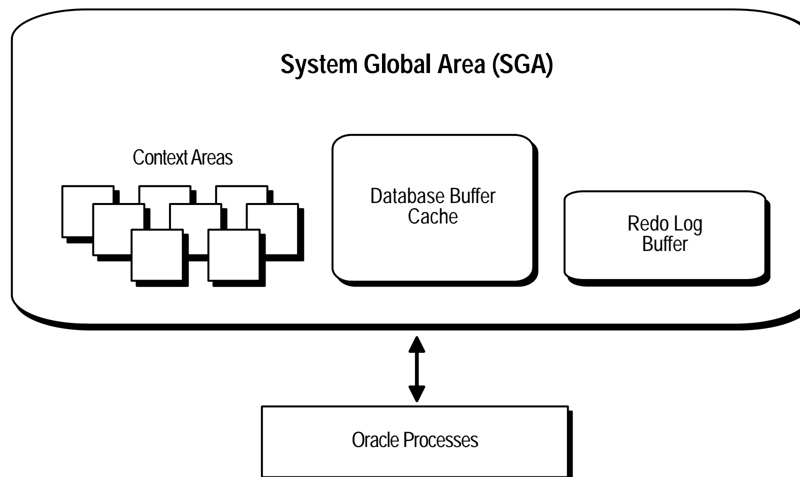


Figure 9 – 1 An Oracle Instance

Oracle starts an instance, then mounts a database to the instance. Multiple instances can execute concurrently on the same machine, each accessing its own physical database. In clustered and massively parallel systems, the Oracle Parallel Server allows a single database to mount multiple instances. When you use Trusted Oracle in OS MAC mode, a single instance can mount multiple databases.

The remaining sections of this chapter explain the memory and process structures and configurations associated with an Oracle instance. See *Oracle7 Parallel Server Concepts & Administration* for more information about the Oracle Parallel Server. See the *Trusted Oracle7 Server Administrator's Guide* for more information about Trusted Oracle.

Process Structure

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.

The process structure of Oracle is important because it defines how multiple activities can occur and how they are accomplished. For example, two goals of a process structure might be

- to simulate a private environment for multiple processes to work simultaneously, as though each process has its own private environment
- to allow multiple processes to share computer resources, which each process needs, but no process needs for long periods of time

The Oracle's process architecture is designed to maximize performance.

Single–Process Oracle Instance

Single–process Oracle (also called single–user Oracle) is a database system in which all Oracle code is executed by one process. Different processes are not used to separate execution of the parts of Oracle and the client application program. Instead, all code of Oracle and the single user's database application is executed by a single process.

Figure 9 – 2 shows a single–process Oracle instance. The single process executes all code associated with the database application **and** Oracle.

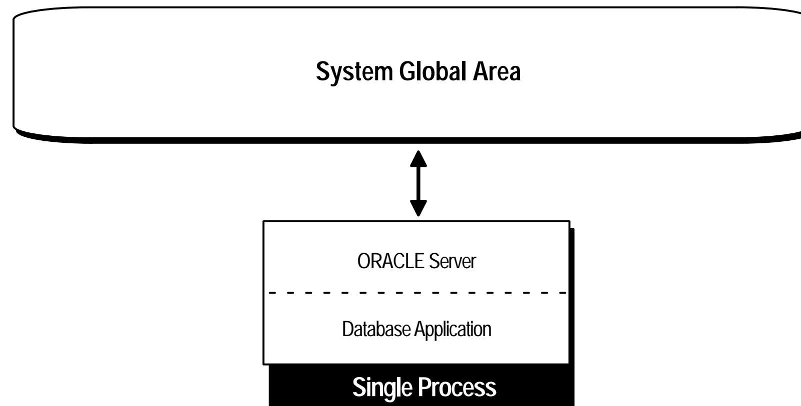


Figure 9 – 2 A Single–Process Oracle Instance

Only one user can access an Oracle instance in a single–process environment; multiple users cannot access the database concurrently. For example, Oracle running under the MS–DOS operating system on a PC can only be accessed by a single user because MS–DOS is not capable of running multiple processes.

Multiple–Process Oracle Instance

Multiple–process Oracle (also called multi–user Oracle) uses several processes to execute different parts of Oracle, and a separate process for each connected user. Each process in a multiple–process Oracle instance performs a specific job. By dividing the work of Oracle and database applications into several processes, multiple users and applications can simultaneously connect to a single database instance while the system maintains excellent performance. Most database systems are multi–user, because one of the primary benefits of a database is managing data needed by multiple users at the same time.

Figure 9 – 3 illustrates a multiple–process Oracle instance. Each connected user has a separate user process and several background processes are used to execute Oracle. This figure might represent multiple concurrent users running an application on the same machine as Oracle; this particular configuration is usually on a mainframe or minicomputer.

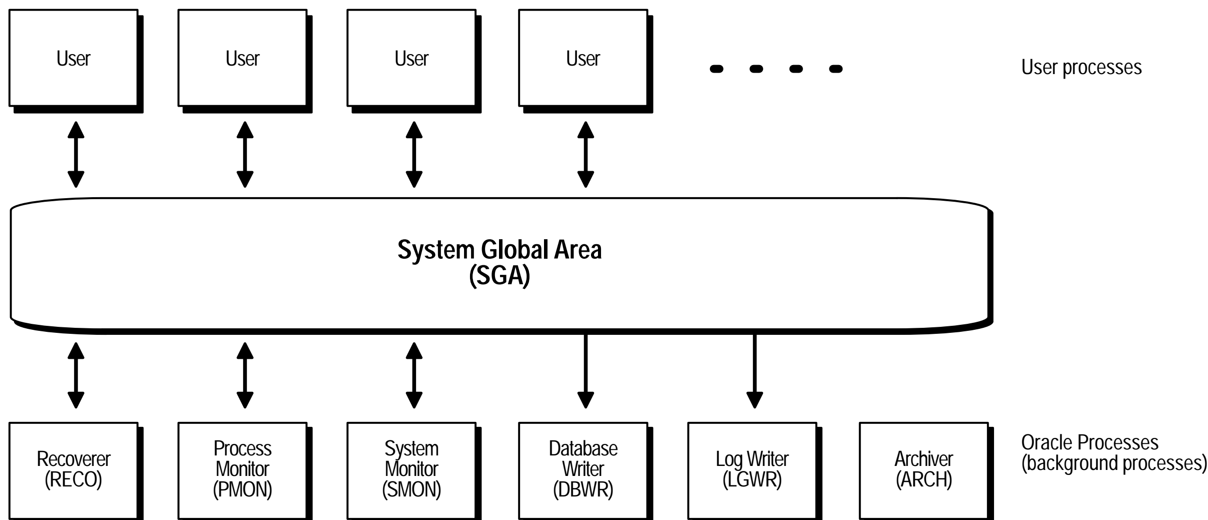


Figure 9 – 3 A Multiple–Process Oracle Instance

In a multiple–process system, processes can be categorized into two groups: user processes and Oracle processes. The following sections explain these classes of processes.

User Processes

When a user runs an application program, such as a Pro*C program, or an Oracle tool, such as Server Manager, Oracle creates a user process to run the user's application.

Oracle Processes

In multiple–process systems, two types of processes control Oracle: server processes and background processes.

Oracle creates server processes to handle the requests of user processes connected to the instance. Often, when the application and Oracle operate on the same machine rather than over a network, a user process and its corresponding server process are combined into a single process to reduce system overhead. However, when the application and Oracle operate on different machines, a user process communicates with Oracle via a separate server process. See "Variations in Oracle Configuration" on page 9–29 for more information.

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 disk (datafiles) 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

To maximize performance and accommodate many users, a multi–process Oracle system uses some additional Oracle processes called *background processes*.

OSDoc

Additional Information: On many operating systems, background processes are created automatically when an instance is started. On other operating systems, the server processes are created as a part of the Oracle installation. See your Oracle operating system–specific documentation for details on how these processes are created.

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 (DBWR)
- Log Writer (LGWR)
- Checkpoint (CKPT)
- System Monitor (SMON)
- Process Monitor (PMON)
- Archiver (ARCH)
- Recoverer (RECO)
- Lock (LCK*n*)
- Snapshot Refresh (SNP*n*)
- Dispatcher (D*nnn*)
- Server (S*nnn*)

Figure 9 – 4 illustrates each background process's interaction with the different parts of an Oracle database, and the following sections describe each process. The Parallel Server is not illustrated; see *Oracle7 Parallel Server Concepts & Administration*.

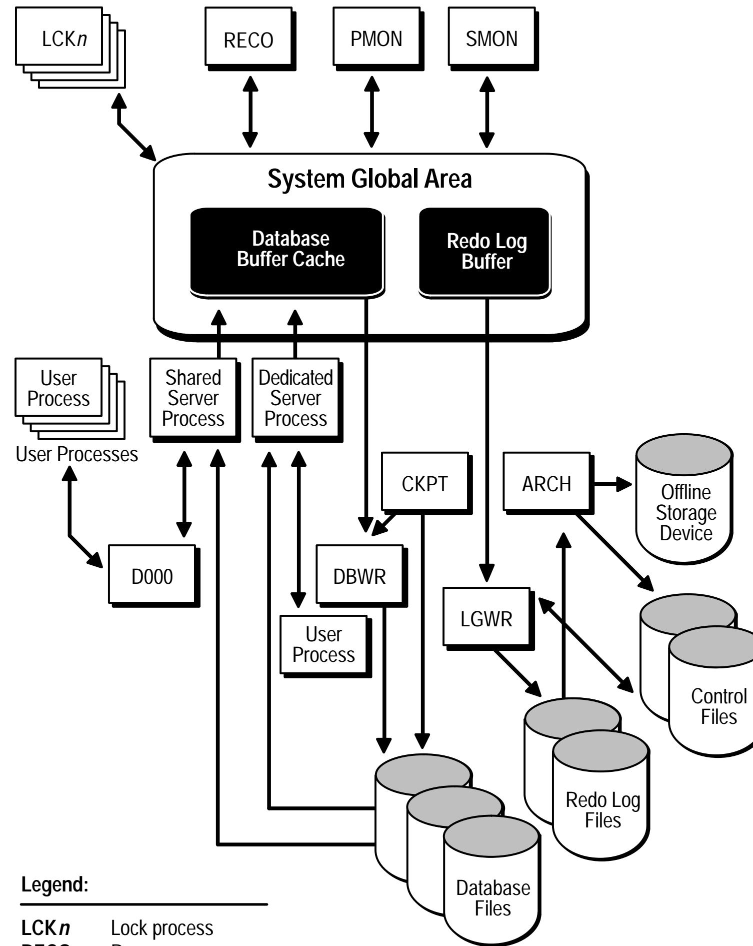


Figure 9 – 4 The Background Processes of a Multiple–Process Oracle Instance

Database Writer (DBWR) *Database Writer process (DBWR)* writes buffers to datafiles. DBWR is an Oracle background process responsible for buffer cache management. For more information about the database buffer cache, see "The Database Buffer Cache" on page 9–17.

When a buffer in the buffer cache is modified, it is marked "dirty". The primary job of the DBWR process is to keep the buffer cache "clean" by writing dirty buffers to disk. As buffers are filled and 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. DBWR manages the buffer cache so that user processes can always find free buffers.

An LRU (least recently used) algorithm keeps the most recently used data blocks in memory and thus minimizes I/O. The database writer process (DBWR) keeps blocks that are used often, for example, blocks that are part of frequently accessed small tables or indexes, in the cache so that they do not need to be read in again from disk. To make room in the buffer cache for other blocks, DBWR removes blocks that are accessed infrequently (for example, blocks that are part of very large tables or leaf blocks from very large indexes) from the system global area (SGA). For information about leaf blocks, see "The Internal Structure of Indexes" on page 5–21.

The LRU scheme causes more frequently accessed blocks to stay in the buffer cache so that when a buffer is written to disk, it is unlikely to contain data that may be useful soon. However, if the DBWR process becomes too active, it may write blocks to disk that are about to be needed again.

The buffer cache has multiple LRU latches. Latches are automatic internal locks that protect shared data structures. The initialization parameter DB_BLOCK_LRU_LATCHES controls how many latches are configured and by default is set to the number of CPUs on your system. This is usually a good value to reduce latch contention for the DBWR processes, thus improving performance.

For More Information

See "Locking Mechanisms" in Chapter 10 of *Oracle7 Server Tuning*.

The DBWR process writes dirty buffers to disk under the following conditions:

- When a server process moves a buffer to the dirty list and discovers that the dirty list has reached a threshold length, the server process signals DBWR to write.
- When a server process searches a threshold limit of buffers in the LRU list without finding a free buffer, it stops searching and signals DBWR to write (because not enough free buffers are available and DBWR must make room for more).
- When a time–out occurs (every three seconds), DBWR signals itself.
- When a checkpoint occurs, the Log Writer process (LGWR) signals DBWR.

In the first two cases, DBWR writes the blocks on the dirty list to disk with a single *multiblock write*. The number of blocks written in a multiblock write varies by operating system.

A *time–out* occurs if DBWR is inactive for three seconds. In this case, DBWR searches a specified number of buffers on the LRU list and writes any dirty buffers that it finds to disk. Whenever a time–out occurs, DBWR searches a new set of buffers. If the database is idle, DBWR eventually writes the entire buffer cache to disk.

When a *checkpoint* occurs, the Log Writer process (LGWR) specifies a list of modified buffers that must be written to disk. DBWR writes the specified buffers to disk. For more information about checkpoints, see “Checkpoints” on page 22–9.

OSDoc

Additional Information: On some platforms, an instance can have multiple DBWRs. In such a case, if one DBWR blocks during a write to one disk, the others can continue writing to other disks. The parameter DB_WRITERS controls the number of DBWR processes. See your Oracle operating system–specific documentation for information about DBWR on your platform.

For more information about DBWR and how to monitor and tune the performance of DBWR, see the *Oracle7 Server Administrator's Guide* and *Oracle7 Server Tuning*.

Log Writer (LGWR) The *Log Writer process (LGWR)* writes the redo log buffer to a redo log file on disk. LGWR is an Oracle background process responsible for redo log buffer management. LGWR writes all redo entries that have been copied into the buffer since the last time it wrote. LGWR writes one contiguous portion of the buffer to disk. LGWR writes

- a commit record when a user process commits a transaction
- redo buffers every three seconds
- redo buffers when the redo log buffer is one–third full
- redo buffers when the DBWR process writes modified buffers to disk

Note: 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 can continue to write to other files in the group (an error is also logged 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. See “The Online Redo Log” on page 22–6 for more information about mirrored

online redo logs and how LGWR functions in this configuration.

The redo log buffer (see "The Redo Log Buffer" on page 9–19) 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.

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.

Oracle uses a "fast commit" mechanism; when a user issues a COMMIT statement, LGWR puts a commit record immediately in the redo log buffer, but the corresponding data buffer changes are deferred until it is more efficient to write them to the datafiles. The atomic write of the redo entry containing the commit record for a transaction is the single event that determines the transaction has committed (then Oracle returns a success code to the committing transaction).

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 Parallel Server configurations and distributed databases. See *Oracle7 Parallel Server Concepts & Administration* and the *Oracle7 Server Administrator's Guide* for more information about SCNs and how they are used.

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. As this happens, other users issue a COMMIT statement. 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, averaging less than one write per COMMIT.

If the CKPT background process is not present, LGWR is also responsible for recording checkpoints as they occur in every datafile's header. See "Checkpoint (CKPT)" below for more information about this background process.

Checkpoint (CKPT) When a checkpoint occurs, Oracle must update the headers of all datafiles to indicate the checkpoint. In normal situations, this job is performed by LGWR. However, if checkpoints significantly degrade system performance (usually, when there are many datafiles), you can enable the Checkpoint process (CKPT) to separate the work of performing a checkpoint from other work performed by LGWR, the Log Writer process (LGWR).

For most applications, the CKPT process is not necessary. If your database has many datafiles and the performance of the LGWR process is reduced significantly during checkpoints, you may want to enable the CKPT process.

The CKPT process does not write blocks to disk; DBWR always performs that work. The statistic *DBWR checkpoints* displayed by the System_Statistics monitor in Server Manager indicates the number of checkpoint messages completed, regardless of whether the CKPT process is enabled or not. See the *Oracle7 Server Administrator's Guide* for information about the effects of changing the checkpoint interval.

The initialization parameter CHECKPOINT_PROCESS enables and disables the CKPT process; its default is FALSE.

> **Note:** See *Oracle7 Parallel Server Concepts & Administration* for additional information about CKPT in an Oracle Parallel Sever.

System Monitor (SMON) The *System Monitor process (SMON)* performs instance recovery at instance start up. SMON is also responsible for cleaning up temporary segments that are no longer in use; it also coalesces contiguous free extents to make larger blocks of free space available. In a Parallel Server environment, SMON performs instance recovery for a failed CPU or instance; see *Oracle7 Parallel Server Concepts & Administration* for more information about SMON in an Oracle Parallel Server.

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.

Process Monitor (PMON) The *Process Monitor (PMON)* performs process recovery when a user process fails. PMON is responsible for cleaning up the cache and freeing resources that the 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 also periodically checks the status of dispatcher and server processes, and restarts any that have died (but not any that Oracle has killed intentionally).

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 (RECO) The *Recoverer process (RECO)* is a process used with the distributed option that automatically resolves failures involving distributed transactions. The RECO background process of a node automatically connects to other databases involved in an in–doubt distributed transaction. When the RECO process re–establishes a connection between involved database servers, it automatically resolves all in–doubt transactions.

The RECO process automatically removes rows corresponding to any resolved in–doubt transactions from each database's pending transaction table.

If the RECO background process attempts to establish communication with a remote server, and the remote server is not available or the network connection has not been re–established, 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.

For more information about distributed transaction recovery, see *Oracle7 Server Distributed Systems, Volume I.*

The RECO background process of an instance is only present if the system permits distributed transactions and if the DISTRIBUTED_TRANSACTIONS parameter is greater than zero. If this parameter is zero, RECO is not created during instance startup.

Archiver (ARCH) The *Archiver process (ARCH)* copies online redo log files to a designated storage device once they become full. ARCH is present only when the redo log is used in ARCHIVELOG mode **and** automatic archiving is enabled. For information on archiving the online redo log, see Chapter 22, "Recovery Structures".

OSDoc

Additional Information: Details of using ARCH are operating system specific; for more information, see Oracle operating system–specific documentation.

Lock (LCK*n*) With the Parallel Server option, up to ten Lock processes (LCK0, . . ., LCK9) provide inter–instance locking. However, a single LCK process (LCK0) is sufficient for most Parallel Server systems. See *Oracle7 Parallel Server Concepts & Administration* for more information about this background process.

Snapshot Refresh (SNP*n*) With the distributed option, up to ten Snapshot Refresh processes (SNP0, ..., SNP9) 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 Snapshot Refresh process is used, the processes share the task of refreshing snapshots.

Dispatcher Processes (D*nnn*) The *Dispatcher processes* allow user processes to share a limited number of server processes. Without a dispatcher, each user process requires one dedicated server process. However, with the multi–threaded server, fewer shared server processes are required for the same number of users. Therefore, in a system with many users, 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:** The multi–threaded server requires SQL*Net Version 2 or later. Each user process that connects to a dispatcher must do so through SQL*Net, 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. This listener process is not part of an Oracle instance; rather, it is part of the networking processes that work with Oracle. See your SQL*Net documentation for more information about the network listener.

When an instance starts, the listener opens and establishes a communication pathway through which users connect to Oracle. Then, each dispatcher gives the listener an address at which the dispatcher listens for connection requests. When a user process makes a connection request, the listener process examines the request and determines if the user can use a dispatcher. If so, the listener process returns the address of the dispatcher process with the lightest load and the user process directly connects to the dispatcher.

Some user processes cannot communicate with the dispatcher (such as users connected using pre–Version 2 SQL*Net) and the network listener process cannot connect such users to a dispatcher. In this case, the listener creates a dedicated server and establishes an appropriate connection.

Trace Files, the ALERT File, and Background Processes

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. See *Oracle7 Server Messages*.

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 Snapshot Refresh processes.

Trace file information can also provide information for tuning applications or an instance. Background processes always write to a trace file when appropriate. However, trace files are written on behalf of server processes (in addition to being written to when there is an internal error) only if the initialization parameter SQL_TRACE is set to TRUE. Regardless of the current value for this parameter, each session can enable or disable trace logging on behalf of the associated server process by using the SQL command ALTER SESSION with the SQL_TRACE parameter. For example, the following statement enables writing to a trace file for the session:

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

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 CREATE/ALTER/DROP DATABASE/TABLESPACE/ROLLBACK SEGMENT SQL statements and STARTUP, SHUTDOWN, ARCHIVE LOG, and RECOVER Server Manager statements
- 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 log of these special operations as an alternative to displaying such information on an operator's console (although many systems display information on the console). If an operation is successful, a message is written in the ALERT file as "completed" along with a timestamp.

Oracle Memory Structures

Oracle uses memory to store the following information:

- program code being executed
- information about a connected session, even if it is not currently active
- data 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 information that is also permanently stored on peripheral memory (for example, a data block)

The basic memory structures associated with Oracle include:

- software code areas
- the system global area (SGA)
  - the database buffer cache
  - the redo log buffer
  - the shared pool
- program global areas (PGA)
  - stack areas
  - data areas
- sort areas

The following topics are included in this section:

- Virtual Memory
- Software Code Areas
- System Global Area (SGA)
- Program Global Area (PGA)
- Sort Areas

Virtual Memory

On many operating systems, Oracle takes advantage of *virtual memory.* Virtual memory is 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.

Note: Usually, it is best to keep the entire SGA in real memory.

Software Code Areas

Software code areas are portions of memory used to store code that is being or may be executed. The code for Oracle is stored in a software area, which is typically at a location different from users' programs — a more exclusive or protected location.

Size of Software Areas

Software areas are usually static in size, only changing when software is updated or reinstalled. The size required varies by operating system.

Read–Only, Shared and Non–Shared

Software areas are read–only and may be installed shared or non–shared. 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 non–shared. 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.

OSDoc

Additional Information: Installing software shared is not an option for all operating systems; for example, it is not on PCs operating MS DOS. See your Oracle operating system–specific documentation for more information.

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, the data in the instance's SGA is "shared" among the users. Consequently, the SGA is often referred to as either the "System Global Area" or the "Shared Global Area".

As described in "An Oracle Instance" on page 9–2, an SGA and Oracle processes constitute an Oracle instance. Oracle automatically allocates memory for an SGA when you start an instance and the memory is reclaimed when you shut down the instance. Each instance has its own SGA.

The SGA is a shared memory area; all users connected to a multiple–process database instance may use information contained within the instance's SGA. The SGA is also writable; several processes write to the SGA during execution of Oracle.

The SGA contains the following subdivisions:

- the database buffer cache
- the redo log buffer
- the shared pool
- request and response queues (if using the multi–threaded server)
- the data dictionary cache
- other miscellaneous information

The Database Buffer Cache

The database buffer cache is a portion of the SGA that holds copies of the data blocks read from datafiles. All user processes concurrently connected to the instance share access to the database buffer cache.

With Release 7.3, the 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 Buffer Cache The buffers in the cache are organized in two lists: the dirty list and the least–recently–used (LRU) list. The *dirty list* holds dirty buffers. A *dirty buffer* is a buffer 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 dirty list. *Free buffers* are buffers that have not been modified and are available for use. *Pinned buffers* are buffers that 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.

When a user process needs to access a block that is not already in the buffer cache, the process must read the block from a datafile on disk into a buffer in the cache. Before reading a 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.

As a user process searches the LRU list, it may find dirty buffers. If the user process finds a dirty buffer, it moves the buffer to the dirty list and continues to search. When a user process finds a free buffer, it reads the block into the buffer and moves it 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 DBWR process to write some of the dirty buffers to disk. See "Database Writer (DBWR)" on page 9–7 for more information about the DBWR background process.

Size of the 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 (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 first time an Oracle user process accesses a piece of data, the process must copy the data from disk to the cache before accessing it. This is called a *cache miss.* When a process accesses a piece of data that is already in the cache, the process can read the data directly from memory. This is called a *cache hit.* Accessing data through a cache hit is faster than data access through a cache miss.

Since the cache has a limited size, all the data on disk cannot fit in the cache. When the cache is full, subsequent cache misses cause Oracle to write data already in the cache to disk to make room for the new data. Subsequent access to the data written to disk results in a cache miss.

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.

The LRU Algorithm and Full Table Scans In one particular case, the user process puts the newly read block's buffer on the LRU end of the list. When the process is performing a full table scan, the blocks for the table are read and put in buffers on the LRU end of the list. This is because a fully scanned table will most likely be needed only briefly, so the blocks should be moved out quickly to leave more frequently used blocks in the cache.

You can prevent the 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 on 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.

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 group on disk. See "Log Writer (LGWR)" on page 9–9 for more information about how the redo log buffer is written to disk.

Size of the Redo Log Buffer The initialization parameter LOG_BUFFER determines the size (in bytes) of 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 is an area in the SGA that contains three major areas: library cache, dictionary cache, and control structures. The following figure shows the contents of the shared pool.

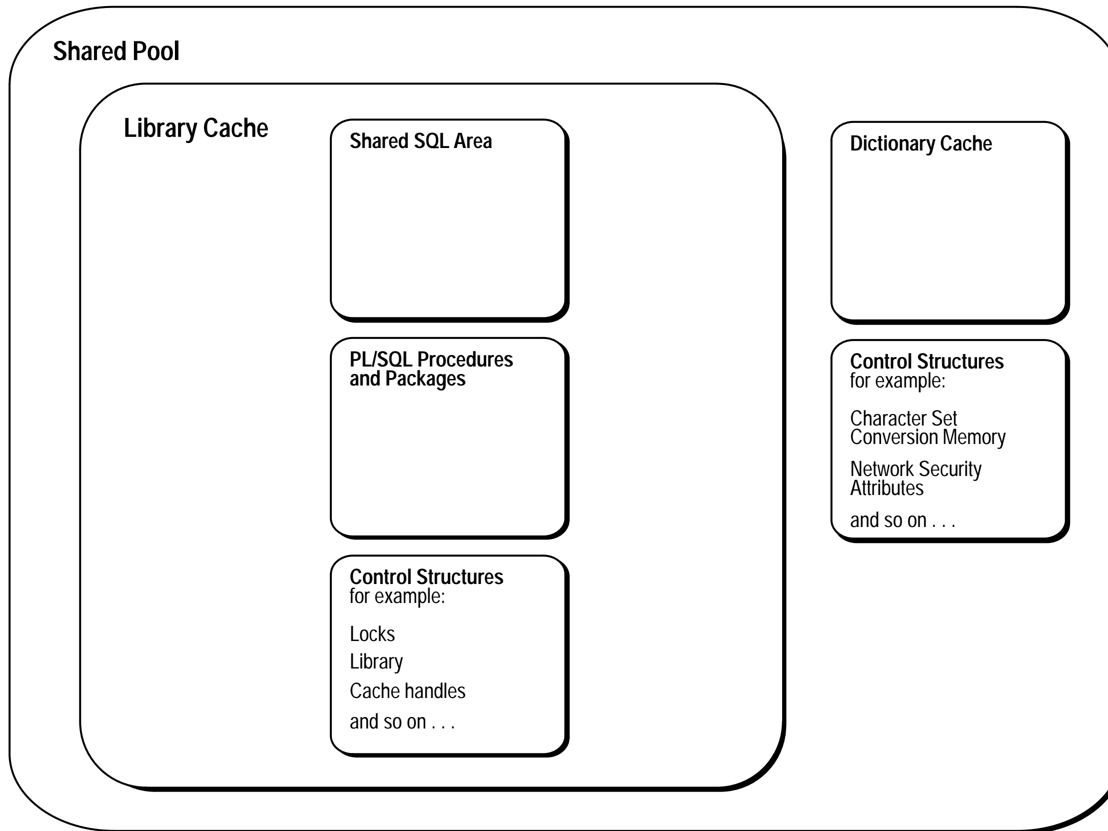


Figure 9 – 5 Contents of the Shared Pool

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

Library Cache

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

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 same shared part for those users. However,

each user must have a separate copy of the statement's private SQL area.

A shared SQL area is a memory area that contains the parse tree and execution plan for a single SQL statement. Oracle allocates memory from the shared pool when a SQL statements is parsed and the size of this memory depends on the complexity of the statement. A shared SQL area is always in the shared pool and is shared for identical SQL statements. For more information about the criteria used to determine identical SQL statements, see "Shared SQL" on page 11–6.

A private SQL area is a memory area that 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/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:

persistent 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.

runtime area
: 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.

  The runtime area is created as the first step of an execute request. For INSERT, UPDATE, and DELETE statements, the runtime area is freed after the statement has been executed. For queries, the runtime area is freed only after all rows are fetched or the query is canceled.

A private SQL area continues to exist until the corresponding cursor is closed. Since Oracle frees the runtime area after the statement completes, generally only the persistent area remains waiting. Application developers should close all open cursors that will not be used again to minimize the amount of memory required for users of the application.

For selects processing large amounts of data where sorts are needed, application developers should cancel the query if the client is satisfied with a partial result of a fetch. For example, in an Oracle Office application, a user can select from a list of over sixty templates for creating a mail message. Oracle Office displays the first ten template names and the user chooses one of these templates. The application can continue to try to display more template names, but because the user has chosen a template, the application should cancel the processing of the rest of the query.

The location of a private SQL area varies depending 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.

How the User Process Manages Private 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.

How Oracle Manages Shared SQL Areas Since shared SQL areas must be available to multiple users, 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. Memory allocation for shared pool is determined by the initialization parameter SHARED_POOL_SIZE. The default value for this parameter is 3.5 megabytes.

If a user process tries to allocate a shared SQL area after the entire shared pool has 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 item. If a shared SQL area is deallocated, the associated SQL statement must be reparsed and reassigned to another shared SQL area when it is next executed.

PL/SQL Program Units and the Shared Pool Oracle processes PL/SQL program units (procedures, functions, packages, anonymous blocks, and database triggers) similar to processing 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/her private SQL area, holding values specific to his/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. Among the data stored in the data dictionary are the following:

- names of all tables and views in the database
- names and datatypes of columns in database tables
- privileges of all Oracle users

This information is useful as reference material for database administrators, application designers, and end users alike. Oracle itself accesses the data dictionary frequently during the parsing of SQL statements. This access is essential to the continuing operation of Oracle. See Chapter 8, "The Data Dictionary," for more information on the data dictionary.

Since the data dictionary is accessed so often by Oracle, two special locations in memory are designated to hold dictionary data. One area is called the *data dictionary cache*, also know as the *row cache*. The other area in memory to hold dictionary data is in the library cache. The data dictionary caches are shared by all Oracle user processes.

Allocation and Reuse of Memory in the Shared Pool

In general, any item (shared SQL area or dictionary row) in the shared pool remains present until it is flushed according to a modified LRU algorithm. The memory for items not being regularly used is freed if space is required for new items that must be allocated some space in

the shared pool. By using a modified LRU algorithm, shared pool items that are shared by many sessions can remain in memory as long as they are useful, even if the process that originally created the item is terminated. As a result, the overhead and processing of SQL statements associated with a multi–user Oracle system is kept to a minimum.

When a SQL statement is submitted to Oracle for execution, there are some special steps to consider. On behalf of every SQL statement, 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 there is already a shared SQL area for the statement, it is used for the execution of the subsequent new instances of the statement. Therefore, instead of having multiple shared SQL areas for identical SQL statements, only one shared SQL area exists for multiple identical DML statements, greatly saving memory, particularly when many users execute the same application.

   Alternatively, if there is not a shared SQL area for a statement, a new shared SQL area is allocated 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 (see "Shared SQL Areas and Private SQL Areas" earlier in this chapter).

Some unique circumstances can also cause a shared SQL area to be flushed from the shared pool. When the ANALYZE command is used to update or delete the statistics of a table, cluster, or index, all shared SQL areas that contain statements that reference the analyzed 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 object. Shared SQL areas are also dependent on the objects referenced in the corresponding SQL statement. If a referenced object is modified, the shared SQL area is *invalidated* (marked invalid) and must be reparsed the next time the statement is executed. See Chapter 16, "Dependencies Among Schema Objects", for

more information about the invalidation of SQL statements and dependency issues.

If you change a database's global database name, all information is flushed from the shared pool. If desired, 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.

Cursors 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. Each user session can open any number of 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, the database administrator can alter the OPEN_CURSORS initialization parameters. For more information about cursors, see "Cursors" on page 11–6.

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 utilize shared SQL areas.

Other SGA Information

Information also stored in the SGA includes the following:

- information communicated between processes, such as locking information
- when an instance is running the multi–threaded server, some contents of the program global areas, and the request and response queues are in the SGA (See "Program Global Area (PGA)" on page 9–26, and "Dispatcher Request and Response Queues" on page 9–37.)

The amount of memory dedicated to all shared areas in the SGA can have performance impact; see the *Oracle7 Server Administrator's Guide* for more information.

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 the entire SGA does not fit in real memory and virtual memory is used to store parts of the SGA, 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 primary determinants of the size of the SGA are the parameters found in a database's parameter file. The parameters that most affect SGA size are the following:

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 Server Manager. You can also see the current instance's SGA size using the Server Manager command SHOW and the SGA option. See the *Server Manager's User's Guide* for more information about the Server Manager command SHOW, and the *Oracle7 Server Administrator's Guide* for discussions of the above initialization parameters and how they affect the SGA. See your installation or user's guide for information specific to your operating system.

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.

Program Global Area (PGA)

The Program Global Area (PGA) is a memory region that contains data and control information for a single process (server or background). Consequently, the PGA is referred to as the "Program Global Area" or the "Process Global Area."

Contents of a PGA

A PGA is allocated by Oracle when a user process connects to an Oracle database and a session is created, though this varies by operating system and configuration. See "Connections, Sessions, and User Processes" on page 9–30 for more information about sessions. The contents of a PGA vary, depending on whether the associated instance is running the multi–threaded server. Figure 9 – 6 summarizes these differences.

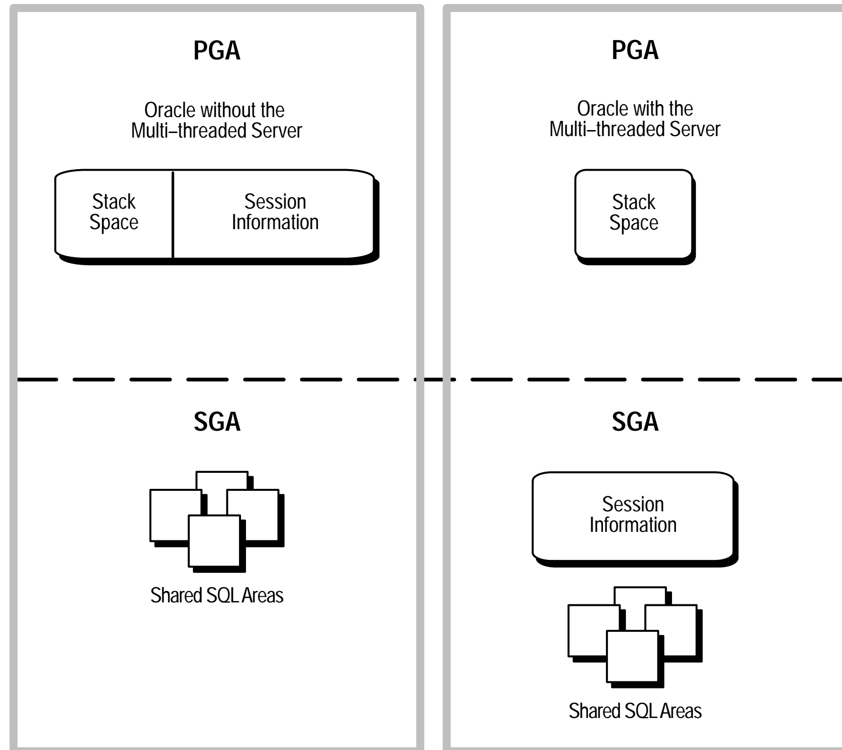


Figure 9 – 6 The Contents of a PGA with and without the Multi–Threaded Server

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 A PGA in an instance running without the multi–threaded server requires additional memory for the user's session, such as private SQL areas and other information. If the instance is running the multi–threaded server, this extra memory is not in the PGA, but is instead allocated in the SGA.

Shared SQL Areas Shared SQL areas are always in shared memory areas of the SGA (not the PGA), with or without the multi–threaded server.

Non–Shared and Writable

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

Size of a PGA

A PGA's size is operating system specific, and not dynamic. 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 error occurs. This error is an Oracle error in an operating system error number range. 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 following initialization parameters affect the sizes of PGAs:

- OPEN_LINKS
- DB_FILES
- LOG_FILES

The size of the stack space in each PGA created on behalf of Oracle background processes (such as DBWR and LGWR) is affected by some additional parameters. See your Oracle operating system–specific documentation for more information.

Sort Areas

Sorting requires space in memory. Portions of memory in which Oracle sorts data are called *sort areas*. A sort area exists in the memory of an Oracle user process that requests a sort. 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 value of this parameter is expressed in bytes. The default value varies depending on your operating system.

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 size to which the sort area is reduced is determined by the initialization parameter SORT_AREA_RETAINED_SIZE. The value of this parameter is expressed in bytes. The minimum value is the equivalent of one database block, the maximum and default value is the value of the SORT_AREA_SIZE initialization 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.

Sort Direct Writes

Sort Direct Writes provides an automatic tuning method for deriving the size and number of direct write buffers based upon the sort area size. The memory for the buffers is taken from the sort area, so only one tuning parameter is necessary. In addition, an optimizer cost model is provided.

If memory and temporary space are abundant on your system and you perform many large sorts to disk, the setting of the initialization parameter SORT_DIRECT_WRITES can increase sort performance.

For Release 7.3 and greater, the default value of SORT_DIRECT_WRITES is AUTO. If the initialization parameter is unspecified or set to AUTO, the database automatically allocates direct write buffers if the SORT_AREA_SIZE is ten times the minimum direct write buffer configuration. In this case, the sort allocates the direct write buffers out of a portion of the total sort area, ignoring the settings of SORT_WRITE_BUFFER_SIZE and SORT_WRITE_BUFFERS.

If you set SORT_DIRECT_WRITES to FALSE, the sorts that write to disk will write through the buffer cache. If you set the parameter to TRUE, each sort allocates additional buffers in memory for direct writes. You can set the initialization parameters SORT_WRITE_BUFFERS and SOFT_WRITE_BUFFER_SIZE to control the number and size of these buffers. The sort writes an entire buffer for each I/O operation. The Oracle process performing the sort writes the sort data directly to the disk, bypassing the buffer cache.

For More Information

See *Oracle7 Server Administrator's Guide.*

Variations in Oracle Configuration

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 an Oracle Forms application), which issues SQL statements to an Oracle database.

Oracle server code — Each user has some Oracle server code executing on his/her behalf, which interprets and processes the application's SQL statements.

In a multiple–process instance, the code for connected users can be configured in one of three variations:

- For each user, both the database application and the Oracle server code are combined in a single user process.
- 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). This configuration is called the *dedicated server architecture*.
- The database application is a different process (a *user* process) than the one that executes the Oracle server code. Furthermore, each server process that executes Oracle server code (a shared server process) can serve multiple user processes. This configuration is called the *multi–threaded server architecture*.

OSSDoc

Additional Information: Some operating systems offer a choice of configurations; see your Oracle operating system–specific documentation for more details on your options. The following sections describe each variation in more detail.

Connections, Sessions, and User Processes

Before describing the variations in Oracle configurations, it is useful to introduce the terms "connection" and "session", which are related to the term "user process" (see "User Processes" on page 9–5), 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 inter–process 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 the 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 concurrently exist for a single Oracle user; for example, a user with the username/password of SCOTT/TIGER can connect to the same Oracle instance several times using the same username.

When the multi–threaded server is not used, a server process is created on behalf of each user session; however, when the multi–threaded server is used, a single server process can be shared among many user sessions; each of the following sections describes the relationship of sessions and processes with respect to the configuration variations.

Oracle Using Combined User/Server Processes

Figure 9 – 7 illustrates one configuration of Oracle. Notice that in this configuration, the database application and the Oracle server code all run in the same process, termed a *user* process.

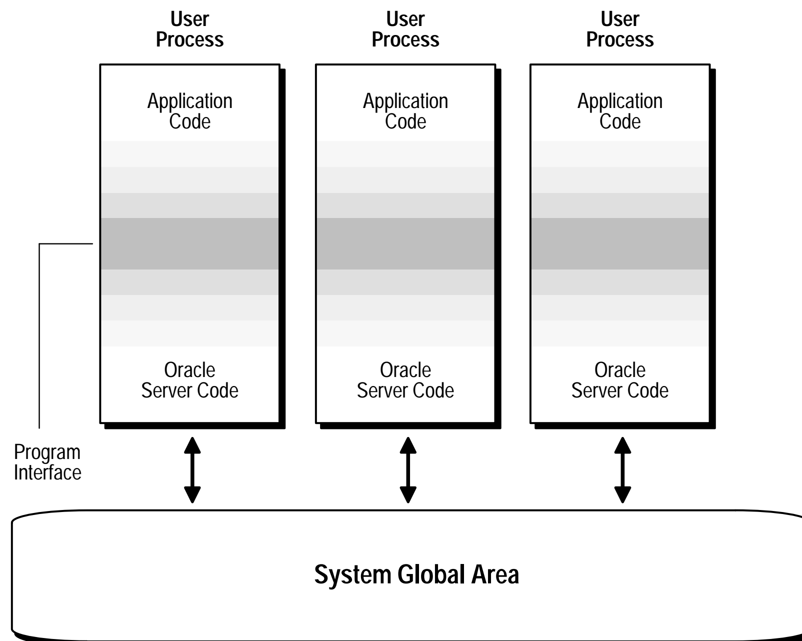


Figure 9 – 7 Oracle Using Combined User/Server Processes

This configuration of Oracle (sometimes called *single–task* Oracle) is only feasible in operating systems that can maintain a separation between the database application and the Oracle code in a single process (such as on the VAX VMS operating system). This separation is required for data integrity and privacy. Some operating systems, such as UNIX, cannot provide this separation and thus must have separate processes run application code from server code to prevent damage to Oracle by the application.

> **Note:** The program interface is responsible for the separation and protection of the Oracle server code and is responsible for passing data between the database application and the Oracle user program. See "The Program Interface" on page 9–41 for more information about this structure.

Only one Oracle connection is allowed at any time by a process using the above configuration. However, in a user–written program it is possible to maintain this type of connection while concurrently connecting to Oracle using a network (SQL*Net) interface.

Oracle Using Dedicated Server Processes

Figure 9 – 8 illustrates Oracle running on two computers using the dedicated server architecture.

Notice that in this type of system, a user process executes the database application on one machine and a server process executes the associated Oracle server on another machine. These two 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.

In this configuration (sometimes called *two–task* Oracle), every user process connected to Oracle has a corresponding dedicated server process. Therefore, there is a one–to–one ratio between the number of user processes and server processes in this configuration. 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).

The dedicated server architecture of Oracle allows client applications being executed on client workstations to communicate with another computer running Oracle across a network. This is illustrated in Figure 9 – 8. However, this configuration of Oracle is also used if the same computer executes both the client application and the Oracle server code, but the host operating system cannot maintain the separation of the two programs if they were to be run in a single process. A common example of such an operating system is UNIX.

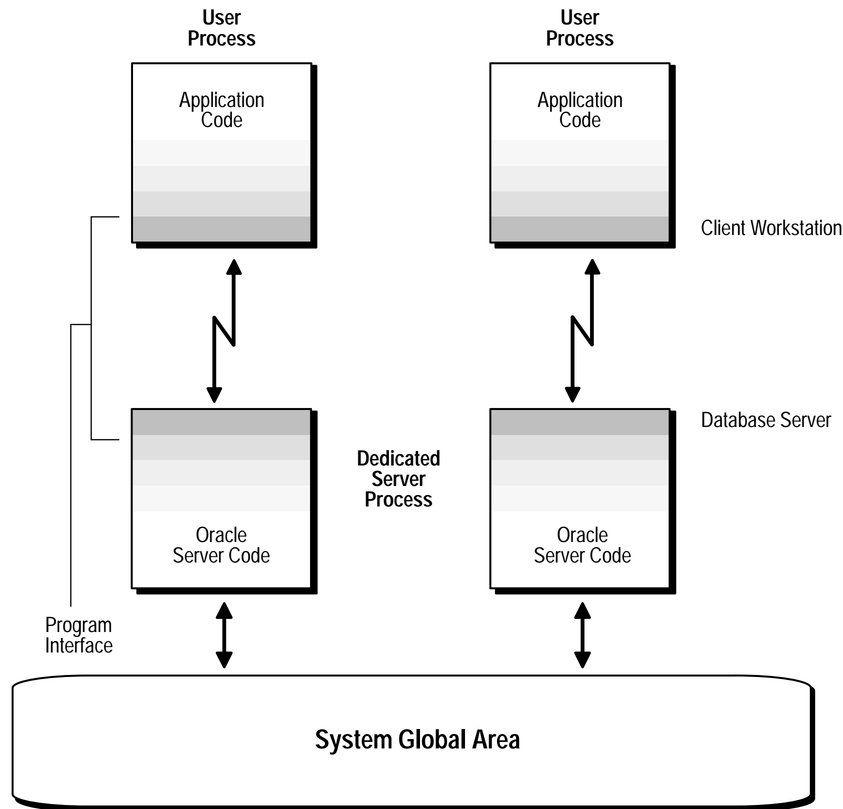


Figure 9 – 8 Oracle Using Dedicated Server Processes

The program interface allows the communication between the two programs. In the dedicated server configuration, communications between the user and server processes is accomplished using different mechanisms:

- If the system is configured so that the user process and the dedicated server process are run by the same computer, the program interface uses the host operating system's inter–process communication mechanism to perform its job.
- If the user process and the dedicated server process are executed by different computers, the program interface also encompasses the communication mechanisms, such as the network software and SQL*Net, between the programs.

OSDoc

Additional Information: These communications links are operating system and installation dependent; see your Oracle operating system–specific documentation and the SQL*Net documentation for more information.

See “The Program Interface” on page 9–41 for additional information about the program interface.

The Multi–Threaded Server

The multi–threaded server configuration allows many user processes to share very few server processes. Without the multi–threaded server configuration, each user process requires its own dedicated server process; a new server process is created for each client requesting a connection. A dedicated server process remains associated to the user process for the remainder of the connection. With the multi–threaded server configuration, many user processes connect to a dispatcher process. The dispatcher routes client requests to the next available shared server process. The advantage of the multi–threaded server configuration is that system overhead is reduced, so the number of users that can be supported is increased.

Contrasting Dedicated Server Processes and Shared Server Processes

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 on the telephone talking to the customer and 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.

The multi–threaded server configuration eliminates the need for a dedicated server process for each connection. A small number of shared server processes can perform the same amount of processing as many dedicated server processes. Also, the amount of memory required for each user is relatively small. Because less memory and process management are required, more users can be supported.

An Overview of the Multi–Threaded Server Architecture

The following types of processes are needed in a system using the multi–threaded server architecture:

- a network listener process that connects user processes to dispatchers and dedicated servers (this process is part of SQL*Net, not Oracle).
- one or more dispatcher processes
- one or more shared server processes

The network listener process waits for incoming connection requests and determines if each user process can use a shared server process. If so, the listener gives the user process the address of a dispatcher process. If the user process requests a dedicated server, the listener process creates a dedicated server process and connects the user process to it. (At least one dispatcher process must be configured and started per network protocol that the database clients will use.)

Note: To use shared servers, a user process must connect through SQL*Net, even if the user process is on the same machine as the Oracle instance.

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 in the SGA, where it is picked up by the next available shared server process. The shared server processes make all the necessary calls to the database to complete each user process's request. When the server completes the request, the server returns the results to the response queue of the dispatcher that the user is connected to the SGA. The dispatcher then returns the completed request to the user process.

In the order entry system example, each clerk's user process connects to a dispatcher; each request made by a clerk is sent to a 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, not making a request to the database, another clerk can use the same shared server process.

Figure 9 – 9 illustrates how user processes communicate with the dispatcher across the two–task interface and how the dispatcher communicates users' requests to shared server processes.

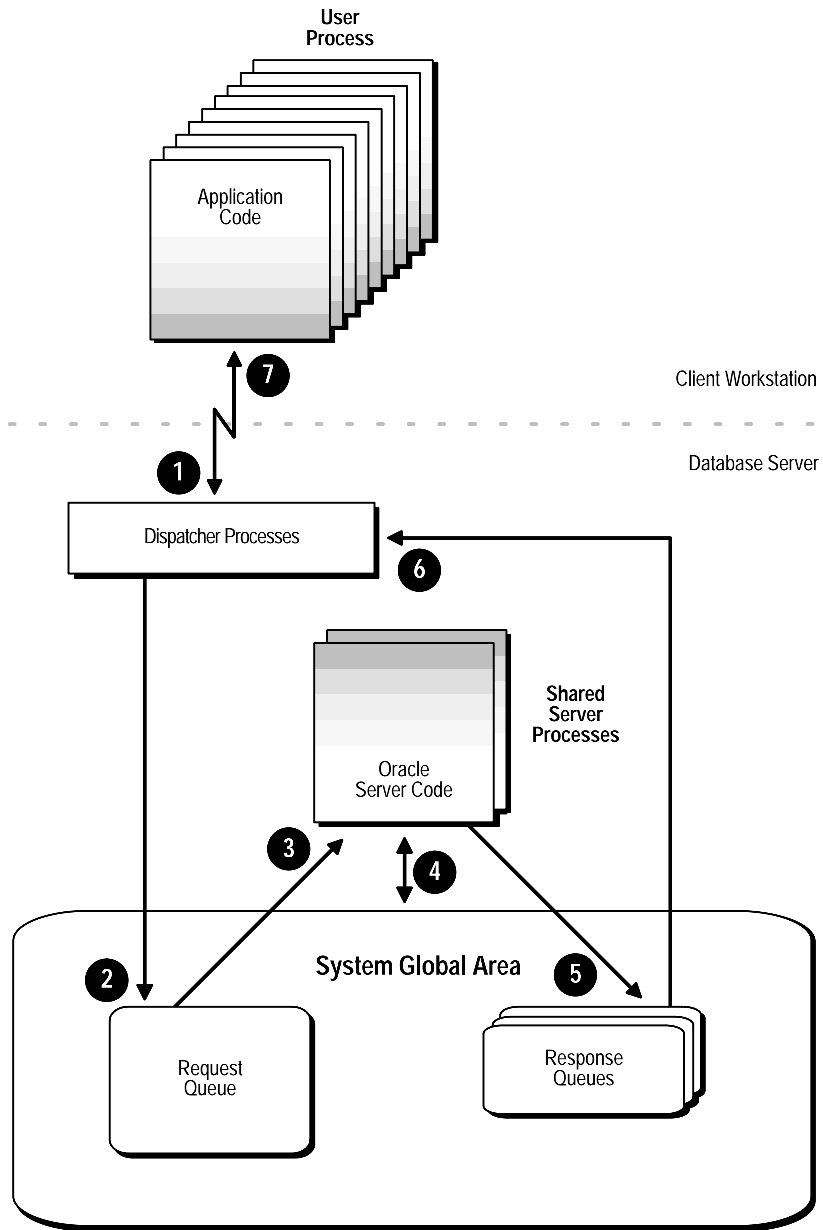


Figure 9 – 9 The Oracle Multi–Threaded Server Configuration and Shared Server Processes

Shared Server Processes

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

The PGA of a shared server process does not contain user–related data; such information needs to be accessible to all shared server processes. The PGA of a shared server process contains only stack space and process–specific variables. "Program Global Area (PGA)" on page 9–26 provides more information about the content of a PGA in different types of instance configurations.

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. See Chapter 17, "Database Access," for more information about resource limits and profiles.

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 initialization parameters MTS_SERVERS and MTS_MAX_SERVERS.

Dispatcher Request and Response Queues

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. The shared server process then places the response on the calling dispatcher's *response queue.* Each dispatcher has its own response queue in the SGA and each dispatcher is responsible for sending completed requests back to the appropriate user process.

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 LOCK TABLE statement.
2. The shared server process used to process 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 users 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 as INTERNAL. When you want to connect as INTERNAL in system configured with multi–threaded servers, you must state in your connect string that you want to use a dedicated server process instead of a dispatcher process (SRVR=DEDICATED).

OSDoc

Additional Information: See your Oracle operating system–specific documentation or SQL*Net documentation for the proper connect string syntax.

Examples of How Oracle Works

Now that the memory structures, processes, and varying configurations of an Oracle database system have been discussed, it is helpful to see how all the parts work together. The following sections demonstrate and contrast the two–task and multi–threaded server Oracle configurations.

An Example of Oracle Using Dedicated Server Processes

The following example is a simple illustration of the dedicated server architecture of Oracle:

1. A database server machine is currently running Oracle using multiple background processes.
2. A client workstation runs a database application (in a user process) such as SQL*Plus. The client application attempts to establish a connection to the server using a SQL*Net driver.
3. The database server is currently running the proper SQL*Net driver. The 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 an identical SQL statement, the server process can use the existing shared SQL area to execute the client's SQL statement.
   - If the shared pool does not contain a shared SQL area for an identical 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 DBWR 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.

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

An Example of Oracle Using the Multi–Threaded Server

The following example is a simple illustration of the multi–threaded server architecture of Oracle:

1. A database server is currently running Oracle using the multi–threaded server configuration.
2. A client workstation runs a database application (in a user process) such as SQL*Forms. The client application attempts to establish a connection to the database server using the proper SQL*Net driver.
3. The database server machine is currently running the proper SQL*Net driver. The 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 SQL*Net Version 2, the Listener informs the user process to reconnect using the address of an available dispatcher process.

   Note: If the user is using SQL*Net Version 1 or 1.1, the SQL*Net listener process creates a dedicated server process on behalf of the user process and the remainder of the example operates as described in the preceding example. (Users using earlier versions of SQL*Net cannot use a shared server process.)

4. The user issues a single SQL statement. For example, the user updates a row into 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. It then processes the SQL statement as described in Steps 5, 6, and 7 of the previous example. Note in Step 5 that parts of the session's private SQL area are created in the SGA.
7. 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.
8. The dispatcher process checks its response queue and sends completed requests back to the user process that made the request.

The Program Interface

The *program interface* is the software layer between a database application and Oracle. The program interface does the following:

- 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 SQL*Net *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.

SQL*Net 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 SQL*Net drivers.

The installation and configuration guide and SQL*Net documentation for your system contains details about choosing and installing drivers and adding new drivers after installation. The SQL*Net documentation describes selecting a driver at runtime while accessing Oracle.

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.

OSDoc

Additional Information: 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 your Oracle operating system–specific documentation for more information about the communication software of your system.

CHAPTER

10

Data Concurrency

A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.

Ralph Waldo Emerson

This chapter explains how Oracle maintains consistent data in a multi–user database environment. The chapter includes:

- Data Concurrency in a Multi–user Environment
- How Oracle Controls Data Concurrency
- How Oracle Locks Data

Data Concurrency in a Multi–user 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 multi–user 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 multi–user database. These concepts are defined here:

data concurrency	Many users can access data at the same time.
data consistency	Users should see 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 the application throughput. Complete isolation of concurrently running transactions could mean that one transaction could not do an insert into a table that was being queried by another. In short, real world considerations usually make it necessary to choose 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.

General Concurrency Issues

The ANSI/ISO SQL standard (SQL92) defines several levels of transaction isolation with differing degrees of impact on transaction processing throughput. These isolation levels are defined in terms of phenomena that must be prevented between concurrently executing transactions.

Preventable Phenomena

The SQL standard defines three phenomena and four levels of isolation that provide protection against the phenomena. The three preventable phenomena are defined as:

dirty reads
: A transaction reads data that has been written by a transaction that has *not* been committed yet.

non–repeatable (fuzzy) reads
: A transaction re–reads 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 satisfy a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

Isolation Levels

The SQL standard defines four levels of isolation in terms of the phenomena a transaction running at a particular isolation level is permitted to experience.

Isolation Level	Dirty Read	Non–Repeatable 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. Read committed is the default and was the only automatic isolation level provided before Release 7.3.

Locking Mechanisms

In general, multi–user databases use some form of data locking to solve the problems associated with data concurrency, integrity, and consistency. *Locks* are mechanisms used to prevent destructive interaction between users 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

Restrictiveness of Locks

In general, you can use two levels of locking in a multi–user database:

exclusive locks
: An exclusive lock prevents the associated resource from being shared and are obtained to modify data. The first transaction to exclusively lock a resource is the only transaction that can alter the resource until the exclusive lock is released.

share locks
: A share lock 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 holds an exclusive lock). Several transactions can acquire share locks on the same resource.

Deadlocks

A *deadlock* is a situation that can occur in multi–user systems that prevents some transactions from continuing work. A deadlock can occur when two or more users are waiting for data locked by each other. Figure 10 – 1 illustrates two transactions in a deadlock.

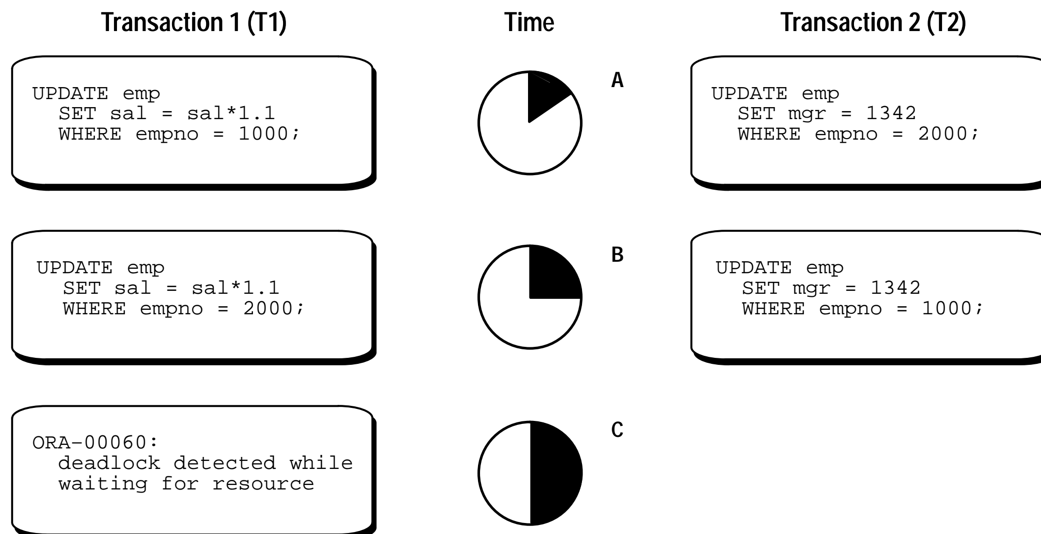


Figure 10 – 1 Two Transactions in a Deadlock

In Figure 10 – 1, 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 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.

Lock Escalation

Lock escalation occurs when numerous locks are held at one level and the database automatically changes the locks to different locks at a higher level. For example, if a single user locks many rows in a table, the database might automatically escalate the user's row locks to a single table lock. With this plan, the number of locks has been reduced, but the restrictiveness of what is being locked has increased.

Lock escalation greatly increases the likelihood of deadlocks. For example, 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 before it can proceed, since the escalator is devoted to T1.

Note: Oracle never escalates locks.

How Oracle Controls Data Concurrency

Oracle maintains data concurrency, integrity, and consistency by using a multiversion consistency model and various types of locks and transactions.

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. Oracle can also provide read consistency to all of the queries in a transaction.

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 10 – 2 shows how Oracle can provide statement–level read consistency using data in rollback segments.

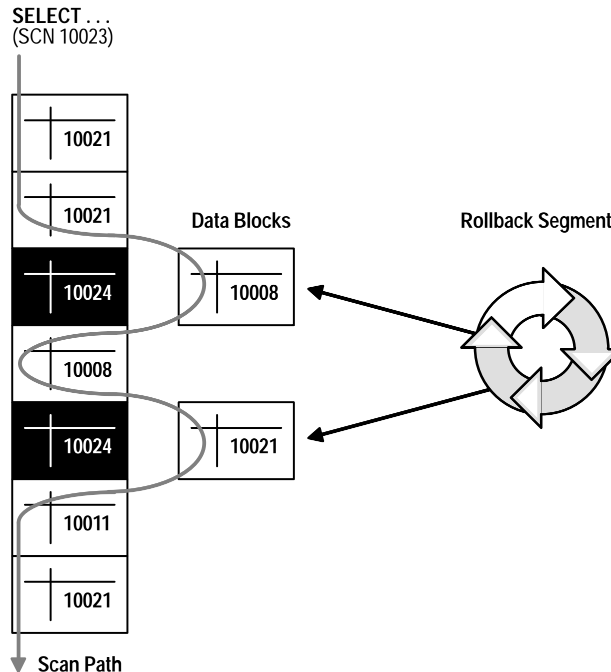


Figure 10 – 2 Transactions and Read Consistency

As a query enters the execution stage, the current system change number (SCN) is determined; in Figure 10 – 2, this system change number is 10023. As data blocks are read on behalf of the query, only blocks written with the observed system change number are used. Blocks with changed data (more recent SCNs) are reconstructed using 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 the data returned by a single query is consistent with respect to 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 query, 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.

While queries used in INSERT, UPDATE, and DELETE statements are guaranteed a consistent set of results, they do not see the changes made by the DML statement itself. In other words, the data the query in these operations sees reflects the state of the data before the operation began to make changes.

Transaction Level Read Consistency

Oracle also allows 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. Therefore, transaction–level read consistency produces repeatable reads and does not expose a query to phantoms.

Oracle Isolation Levels

Oracle provides three transaction isolation modes:

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 non–repeatable 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 see non–repeatable 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 DBAs should set an isolation level for transactions appropriate for the specific application and workload. As an application designer, application developer, or DBA, you can choose different isolation levels for different transactions.

You can set the isolation level of a transaction by using one of these commands 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 command, you can use the ALTER SESSION command to set the transaction isolation level for all subsequent transactions:

```
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;
ALTER SESSION SET ISOLATION_LEVEL READ COMMITTED;
```

You can also change the default transaction isolation level for the system by using the ALTER SYSTEM command. For detailed

information on any of these SQL commands, see chapter 4 of *Oracle7 Server SQL Reference.*

Read Committed Isolation

The default isolation level for Oracle is read committed. This degree of isolation is appropriate for most applications. With read committed isolation levels, Oracle causes each query to execute with respect to its own snapshot time, thereby permitting non–repeatable reads and phantoms for two 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 mode is suitable for environments with large databases and short transactions that update only a few rows. It is appropriate for environments where both of the following are true:

- there is a relatively low chance that two concurrent transactions will modify the same rows
- relatively long–running transactions are primarily read–only

Serializable mode prevents interactions between transactions that would preclude them from executing one at a time. In other words, concurrent transactions executing in serializable mode are only permitted to make database changes the could have made if the transactions had been scheduled to execute one after another. This mode ensures transactions move the database from one consistent state to another consistent state. It prevents potentially harmful interactions between concurrently executing transactions, but causes a reduction in throughput that is often unacceptable.

A serializable transaction executes against the database as it existed at the beginning of the transaction. A serializable transaction cannot modify rows changed by other transactions that are ”too recent,” that is, that commit *after* the serializable transaction began.

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
```

Here is an example:

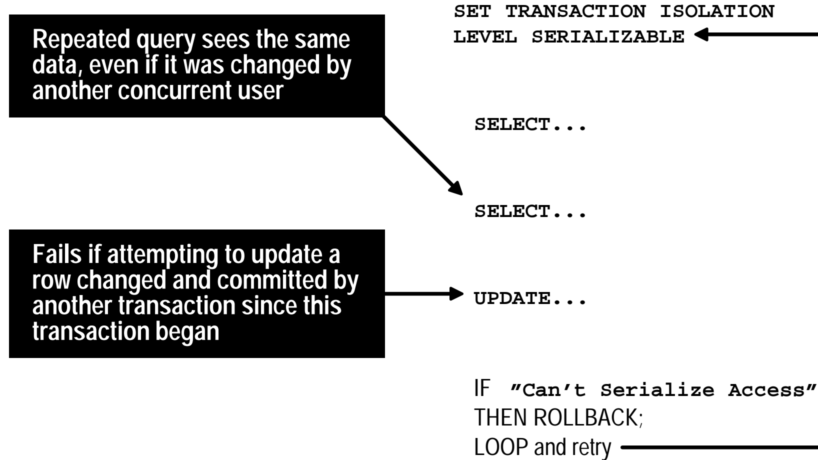


Figure 10 – 3 Serializable Transaction Failure

When a serializable transaction fails with the "Can't 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

Oracle stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to serializable, you must use the ALTER TABLE or CREATE TABLE command to set INITRANS to at least 3.

Additional Considerations for Serializable Isolation

The following sections provide additional background and information useful to application developers and database administrators planning to use serializable transactions.

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 rollback 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, read committed and serializable transactions behave differently if the other (blocking) transaction commits. When the other transaction commits and releases its locks, a read committed

transaction will proceed with its intended update. A serializable transaction, however, will fail with the error "Can't serialize access", since the other transaction has committed a change that was made since the serializable transaction began.

Serializable Transactions and Row Locking

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 within the 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.

Update Intensive Environments and Serializable Transactions

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.

Higher values of INITRANS should be used for tables that will experience many transactions updating the same blocks. This parameter will cause Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block.

Referential Integrity and Serializable Transactions

Because Oracle does not use read locks, even in 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 not change 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.

For more information about referential integrity and serializable transactions, see *Oracle7 Server Application Developer's Guide*.

Oracle Parallel Server and Distributed Transactions

Both read committed and serializable transaction isolation levels can be used with the Oracle Parallel Server (a cluster of several Oracle instances running against a single database across a number of nodes).

Oracle supports distributed serializable transactions, where a given transaction updates data in multiple physical databases (protected by two–phase commit to ensure all nodes or none commit). In a distributed database environment, all servers (whether Oracle or

non–Oracle) that participate in a serializable transaction are required to support that transaction 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 indicating this. In this way, the transaction can rollback 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.

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. The rest of this section compares the two isolation modes and provides information helpful in choosing between them.

A useful way to describe the read committed and serializable isolation levels in Oracle is to consider the following: 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. In an operation that is not transaction set consistent, 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.

Differences Between Read Committed and Serializable Transactions

The table below summarizes key differences between read committed and serializable transactions in Oracle.

	Read committed	Serializable
Dirty write	Not possible	Not possible
Dirty read	Not possible	Not possible
Non–repeatable 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

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. You should choose an isolation level that provides the required degree of consistency while satisfying performance expectations. Frequently, for high performance environments, the choice of isolation levels involves making a tradeoff between consistency and concurrency (transaction throughput).

Both Oracle isolation modes provide high levels of consistency and concurrency (and performance) through the combination of row–level locking and Oracle's multiversion concurrency control system. Because readers and writers don't block one another in Oracle, while queries still see consistent data, both read committed and serializable isolation provide a high level of concurrency for high performance, without the need for reading uncommitted ("dirty") data.

Read committed isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results (due to phantoms and non–repeatable reads) for some transactions. The serializable isolation level provides somewhat more consistency by protecting against phantoms and non–repeatable reads and may be important where a read/write transaction executes a query more than once. However, serializable mode requires applications to check for the "Cannot serialize access" error and can significantly reduce throughput in an environment with many concurrent transactions accessing the same data for update. Application logic that checks database consistency must take into account the fact reads don't block writes in either mode.

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 previous to release 7.3.

Often, high performance environments with high transaction arrival rates require more throughput and faster response times than can be achieved with serializable isolation. On the other hand, an environment that supports few users with a very low transaction arrival rate faces exceedingly low risk of incorrect results due to phantoms and non–repeatable reads. Both of these environments are suitable for read committed isolation.

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 re–issue the same query twice, so for many applications protection against phantoms and non–repeatable reads is not important. Therefore many developers choose read committed to avoid the need to write such error checking and retry code in each transaction.

Choosing Serializable Isolation

Oracle's serializable isolation mode 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.

Unlike other implementations of serializable mode, which lock blocks for read as well as write, Oracle provides the benefit of non–blocking queries and the fine granularity of row–level locking. Oracle's row–level locking and non–blocking sequence generators also reduce write/write contention. For applications that experience mostly read/write contention, Oracle serializable mode can provide significantly more throughput than other systems. Therefore, some applications might be suitable for serializable mode on Oracle but not on other systems.

Because all queries in an Oracle serializable transaction see the database as of a single point in time, this mode 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.

Coding serializable transactions requires extra work by the application developer (to check for the "Cannot serialize access" error and to rollback 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 rollback, 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 frequently occur, 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 may be 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 successfully complete, so the application should be coded to limit the number of retries.

Although Oracle serializable mode is compatible with SQL92 and offers many benefits as 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

The only data 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 only contend for data 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.

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.

For a complete description of the internal locks used by Oracle, see "Types of Locks" on page 10–18.

Transactions and Data Concurrency

Oracle can provide 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 will automatically manage locking.

Duration of Locks

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 only become visible 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 on 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 and 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. For more information about table locks, see "Table Locks"on page 10–20.

Oracle *does not* escalate any locks at any time from one level of granularity (for example, rows) to another (for example, table), reducing the chance of deadlocks.

Deadlock Detection

Oracle automatically detects deadlock situations and resolves them automatically 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, non–distributed 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 "Explicit (Manual) Data Locking" for more information about manually acquiring locks and for an example of a deadlock situation.

Avoiding Deadlocks

Multi–table deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order as each other, 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 certain events occur, and the transaction no longer requires the resource.

> **Note:** While reading this section, 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 explicitly lock any resource. 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.

Later sections also describe situations where you might wish to acquire locks manually or to alter the default locking behavior of Oracle and

explain how you can do so; see "Explicit (Manual) Data Locking" on page 10–29.

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:

data locks (DML locks)	Data locks protect data. For example, *table locks* lock entire tables, *row locks* lock selected rows.
dictionary locks (DDL locks)	Dictionary locks protect the structure of objects. For example, dictionary locks protect 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.

The following sections discuss data locks, dictionary locks, and internal locks, respectively. For more information about distributed and PCM locks, see *Oracle7 Parallel Server Concepts & Administration*.

Data Locks

The purpose of a data lock (DML lock) is to guarantee the integrity of data being accessed concurrently by multiple users. Data locks prevent destructive interference of simultaneous conflicting DML and/or DDL operations. For example, Oracle data 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.

Row Locks (TX)

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 Enterprise Manager. Enterprise Manager might display TM for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

A transaction acquires an exclusive data 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. Row locks are always acquired automatically by Oracle as a result of the statements listed above.

Rows Locks and Table Locks If a transaction obtains a row lock for a row, the transaction also acquires a table lock for the corresponding table. A table lock also must be obtained to prevent conflicting DDL operations that would override data changes in a current transaction. The following section explains table locks, and "DDL Locks (Dictionary Locks)" on page 10–26 explains the locks necessary for DDL operations.

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. (For more information about exclusive DDL locks, see "Exclusive DDL Locks" on page 10–26.)

A table lock can be held in any of several modes: row share (RS), row exclusive (RX), share lock (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 10 – 1 shows the modes of table locks that statements acquire and operations that those locks permit and prohibit.

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 SHARE 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

Table 10 – 1 Summary of Table Locks

RS: row share
SRX: share row exclusive
RX: row exclusive
X: exclusive
S: share
* 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. For more information about manual locking, see "Explicit (Manual) Data Locking" on page 10–29.

Row Share Table Locks (RS) A row share table lock (also sometimes internally called a *sub–share 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 internally called a *sub–exclusive 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 only update the table 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–sub–exclusive 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.

Data 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 data is automatically locked on behalf of different DML operations. Table 10 – 2 summarizes the information in the following sections.

DML Statement	Row Locks?	Mode of Table Lock
SELECT ...FROM table		
INSERT INTO table ...	✓	RX
UPDATE table ...	✓	RX
DELETE FROM table ...	✓	RX
SELECT ... FROM table ... FOR UPDATE OF ...	✓	RS
LOCK TABLE table IN ...		
ROW SHARE MODE		RS
ROW EXCLUSIVE MODE		RX
SHARE MODE		S
SHARE EXCLUSIVE MODE		SRX
EXCLUSIVE MODE		X

RS: row share
RX: row exclusive
S: share
SRX: share row exclusive
X: exclusive

Table 10 – 2 Locks Obtained By DML Statements

Default Locking for Queries Queries are included in the following kinds of statements:

```
SELECT
INSERT . . . SELECT . . . ;
UPDATE . . . ;
DELETE . . . ;
```

They do **not** include the following statements:

```
SELECT . . . FOR UPDATE OF . . . ;
```

Note that INSERT, UPDATE, and DELETE statements can have implicit queries as part of the statement.

Queries are the SQL statements least likely to interfere with other SQL statements because they only read data. 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 *non–blocking 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 Statements 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. Therefore, 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 the 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 (Dictionary Locks)

A DDL lock protects the definition of a schema object (for example, a table) 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 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.

A dictionary lock is acquired automatically by Oracle 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

Certain DDL operations require exclusive DDL locks for a resource to prevent destructive interference with other DDL operations that might modify or reference the same 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.

In addition to DDL locks, DDL operations also acquire DML locks (data locks) on the object to be modified.

Most DDL operations acquire exclusive DDL locks on the object to be modified (except for those listed in the next section, "Share DDL Locks").

During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the object by another operation, the acquisition waits until the older DDL lock is released and then proceeds.

Share DDL Locks

Certain 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 object will remain constant for the duration of the transaction.

A share DDL lock is acquired on an object for DDL statements on the object that include the following commands: 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 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. See Chapter 16, "Dependencies Among Schema Objects", for more information about dependency management. 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 remains in the shared pool.

Duration of DDL Locks

The duration of a DDL lock varies depending 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.

Internal Locks and Latches

Internal locks and latches protect internal database and memory structures. These structures are inaccessible to users, since users have no need for control over their occurrence or duration. The following information will help you interpret the Server Manager 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 are higher–level, more complex mechanisms than latches and serve a variety of purposes. Details on the three categories of internal locks follow.

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 different 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. For more information. see *Oracle7 Parallel Server Concepts & Administration.*

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

In all cases, Oracle automatically performs locking 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 three levels:

transaction level — Transactions including the following SQL statements override Oracle's default locking: the LOCK TABLE command (which locks either a table or, when used with views, the underlying base tables) and the SELECT.. FOR UPDATE command. Locks acquired by these statements are released after the transaction commits or rolls back. For information about each command, see the *Oracle7 Server SQL Reference* .

session level — A session can set the required transaction isolation level with the ALTER SESSION command.

system level — An instance can be started with non–default locking by adjusting the initialization parameter ISOLATION_LEVEL.

> **Note:** If Oracle's default locking is overridden at any level, the database administrator or application developer should be sure that the overriding locking procedures operate correctly. They must satisfy the following criteria: data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or are appropriately handled.

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 is not included, but reads "resource busy and acquire with NOWAIT specified." User–entered text is in **bold**.

Transaction 1	Time Point	Transaction 2
	1	
LOCK TABLE scott.dept IN ROW SHARE MODE; Statement processed	2	**DROP TABLE scott.dept;** DROP TABLE scott.dept * ORA–00054 *(exclusive DDL lock not possible because of T1's table lock)*
	3	**LOCK TABLE scott.dept IN EXCLUSIVE MODE NOWAIT;** ORA–00054
	4	SELECT LOC FROM scott.dept WHERE deptno = 20 FOR UPDATE OF loc; LOC - - - - - - - DALLAS 1 row selected
UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20; *(waits because T2 has locked same rows)*	5	
	6	**ROLLBACK;** *(releases row locks)*
1 row processed. ROLLBACK;	7	
LOCK TABLE scott.dept IN ROW EXCLUSIVE MODE; Statement processed.	8	
	9	**LOCK TABLE scott.dept IN EXCLUSIVE MODE NOWAIT;** ORA–00054

Transaction 1	Time Point	Transaction 2
	10	**LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	11	LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT; ORA-00054
	12	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** 1 row processed.
	13	**ROLLBACK;**
SELECT loc FROM scott.dept WHERE deptno = 20 FOR UPDATE OF loc; LOC - - - - - - - DALLAS 1 row selected.,	14	
	15	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** *(waits because T1 has locked same rows)*
ROLLBACK;	16	
	17	1 row processed. (conflicting locks were re-leased) ROLLBACK;
LOCK TABLE scott.dept IN ROW SHARE MODE Statement processed	18	
	19	**LOCK TABLE scott.dept IN EXCLUSIVE MODE NOWAIT;** ORA-00054

Transaction 1	Time Point	Transaction 2
	20	**LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	21	**LOCK TABLE scott.dept IN SHARE MODE;** Statement processed.
	22	**SELECT loc FROM scott.dept WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	23	**SELECT loc FROM scott.dept WHERE deptno = 20 FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.
	24	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** *(waits because T1 holds conflicting table lock)*
ROLLBACK;	25	
	26	1 row processed. *(conflicting table lock released)* ROLLBACK;
LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE; Statement processed.	27	

Transaction 1	Time Point	Transaction 2
	28	**LOCK TABLE scott.dept IN EXCLUSIVE MODE NOWAIT;** ORA-00054
	29	**LOCK TABLE scott.dept IN SHARE ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	30	**LOCK TABLE scott.dept IN SHARE MODE NOWAIT;** ORA-00054
	31	**LOCK TABLE scott.dept IN ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	32	**LOCK TABLE scott.dept IN SHARE MODE NOWAIT;** ORA-00054
	33	**SELECT loc FROM scott.dept WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.
	34	**SELECT loc FROM scott.dept WHERE deptno = 20 FOR UPDATE OF loc;** LOC - - - - - - DALLAS 1 row selected.
	35	**UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20;** *(waits because T1 holds conflicting table lock)*

Transaction 1	Time Point	Transaction 2
UPDATE scott.dept SET loc = 'NEW YORK' WHERE deptno = 20; *(waits because T2 has locked same rows)*	36	(deadlock)
Cancel operation ROLLBACK;	37	
	38	1 row processed
LOCK TABLE scott.dept IN EXCLUSIVE MODE;	39	
	40	**LOCK TABLE scott.dept IN EXCLUSIVE MODE;** ORA-00054
	41	**LOCK TABLE scott.dept IN ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	42	**LOCK TABLE scott.dept IN SHARE MODE;** ORA-00054
	43	**LOCK TABLE scott.dept IN ROW EXCLUSIVE MODE NOWAIT;** ORA-00054
	44	**LOCK TABLE scott.dept IN ROW SHARE MODE NOWAIT;** ORA-00054
	45	**SELECT loc FROM scott.dept WHERE deptno = 20;** LOC - - - - - - DALLAS 1 row selected.

Transaction 1	Time Point	Transaction 2

	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.

SELECT loc
  FROM scott.dept
  WHERE deptno = 10;
 	
LOC
– – – – – –
(T1 does not see uncommitted data)
	53	

	54	**COMMIT;**

Transaction 1	Time Point	Transaction 2
SELECT loc **FROM scott.dept** **WHERE deptno = 10;** LOC - - - - - - *(same results seen even af-ter 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. For more information about Oracle Lock Management services, see the *Oracle7 Server Application Developer's Guide.*

PART

V

Data Access

CHAPTER

11

SQL and PL/SQL

High thoughts must have high language.

Aristophanes: *Frogs*

This chapter provides an overview of SQL, the Structured Query Language, and PL/SQL, Oracle's procedural extension to SQL. The chapter includes:

- Structured Query Language (SQL)
- PL/SQL

For complete information on PL/SQL, see the *PL/SQL User's Guide and Reference.*

Structured Query Language (SQL)

SQL is a very simple, yet powerful, database access language. SQL is a non–procedural 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 SQL implemented by Oracle Corporation for Oracle is 100% compliant at the Entry Level with the ANSI/ISO 1992 standard SQL data language.

Oracle SQL includes many extensions to the ANSI/ISO standard SQL language, and Oracle tools and applications provide additional commands. The Oracle tools SQL*Plus and Server Manager allow you to execute any ANSI/ISO standard SQL statement against an Oracle database, as well as additional commands 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.

See the *Oracle7 Server SQL Reference* for more information about SQL commands and other parts of SQL (for example, functions) and the *Oracle Server Manager User's Guide* for more information about Server Manager commands, including their distinction from SQL commands.

This section includes the following topics:

- SQL statement
- Identifying Non–Standard SQL
- Recursive SQL
- Cursors
- Shared SQL
- Parsing

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 command.* A statement partially consists 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.

The statement must be the equivalent of a SQL "sentence," as in

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

Only a 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.

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

Each category of SQL statement is briefly described below.

> **Note:** Oracle also supports the use of SQL statements in PL/SQL program units; see Chapter 14, "Procedures and Packages," and Chapter 15, "Database Triggers," for more information about this feature.

Data Manipulation Statements (DML)

DML statements query or manipulate data in existing schema objects. They allow you to do the following:

- Remove rows from tables or views (DELETE).
- See the execution plan for a SQL statement (EXPLAIN PLAN).
- Add new rows of data into a table or view (INSERT).
- Lock a table or view, temporarily limiting other users' access to it (LOCK TABLE).
- Retrieve data from one or more tables and views (SELECT).
- Change column values in existing rows of a table or view (UPDATE).

DML statements are the most frequently used SQL statements. Some examples of DML statements follow:

```
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');
```

Transaction Control Statements

Transaction control statements manage the changes made by DML statements and group DML statements into transactions. They allow you to do the following:

- 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).

Data Definition Statements (DDL)

DDL statements define, alter the structure of, and drop schema objects. DDL statements allow you to do the following:

- 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 follow:

```
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;
```

For specific information on DDL statements that correspond to database and data access, see Chapter 17, "Database Access", Chapter 18, "Privileges and Roles", and Chapter 19, "Auditing".

Session Control Statements

Session control commands manage the properties of a particular user's session. For example, they allow you to do the following:

- 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 commands change the properties of the Oracle Server instance. The only system control command 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 within a procedural language program. They are used with the Oracle Precompilers. Embedded SQL statements allow you to do the following:

- Define, allocate, and release cursors (DECLARE CURSOR, OPEN, CLOSE).
- Declare a database name and connect to Oracle (DECLARE DATABASE, CONNECT).
- Assign variable names, initialize descriptors, and specify how error and warning conditions are handled (DECLARE STATEMENT, DESCRIBE, WHENEVER).
- Parse and execute SQL statements, and retrieve data from the database (PREPARE, EXECUTE, EXECUTE IMMEDIATE, FETCH).

Identifying Non–Standard SQL

Oracle provides features beyond the standard SQL "Database Language with Integrity Enhancement". The Federal Information Processing Standard for SQL (FIPS 127–2) requires a method for identifying SQL statements that use vendor–supplied 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. For information on how to use the FIPS flagger, see the *Oracle7 Server SQL Reference*, the *Programmer's Guide to the Oracle Precompilers*, or the *SQL*Module User's Guide and Reference*.

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 an area in memory in which a parsed statement and other information for processing the statement are kept; such an area is also called a private SQL area. 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.

Shared SQL

Oracle automatically notices when applications send identical SQL statements to the database. If two identical statements are issued, the SQL area used to process the first instance of the statement is *shared*, or used for processing subsequent instances of that same statement.

Therefore, instead of having multiple shared SQL areas for identical SQL statements, 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.

For more information on shared SQL, see the *Oracle7 Server Application Developer's Guide.*

What Is Parsing?

Parsing is one step 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 performs these tasks:

- 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 parses the statement, performing these tasks:

- Oracle generates the parsed representation of the statement.
- 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. Once 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.

Since both parse calls and parsing can be expensive relative to execution, it is desirable to perform them as seldom as possible.

This discussion applies also to the parsing of PL/SQL blocks and the allocation of PL/SQL areas. (See the description of PL/SQL in the next section.) 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.

PL/SQL

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL allows you to mix SQL statements with procedural constructs. PL/SQL provides the capability to 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. For information on stored procedures, functions, packages, and database triggers, see Chapter 14, "Procedures and Packages", and Chapter 15, "Database Triggers".

How PL/SQL Executes

The *PL/SQL engine* is a special component of many Oracle products, including the Oracle Server, that processes PL/SQL. Figure 11 – 1 illustrates the PL/SQL engine contained in Oracle Server.

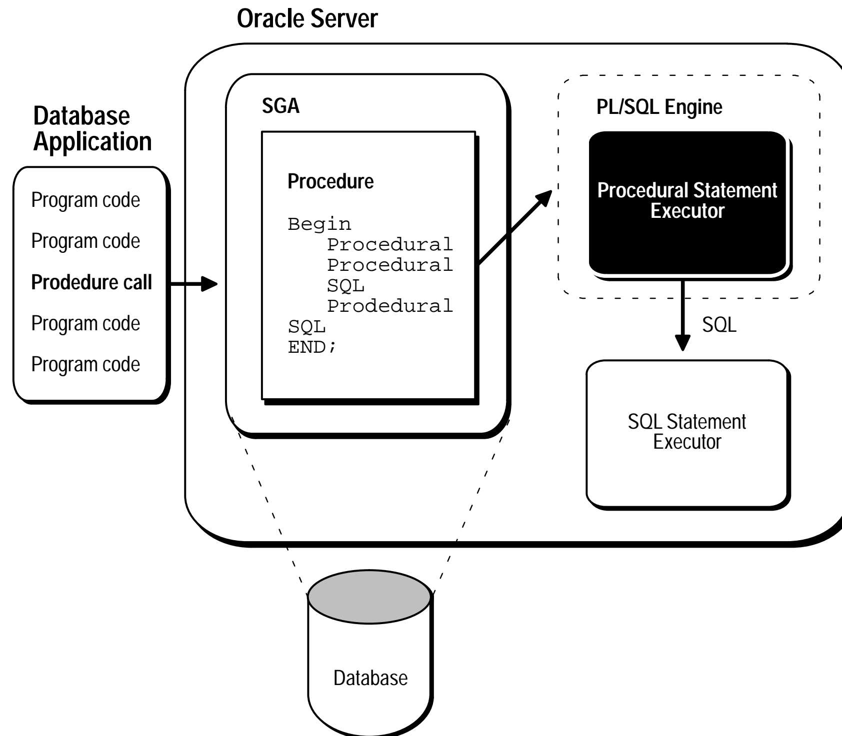


Figure 11 – 1 The PL/SQL Engine and the Oracle Server

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

Language Constructs for PL/SQL

PL/SQL blocks can include the following PL/SQL language constructs:

- variables and constants
- cursors
- exceptions

The following sections give a general description of each construct; see the *PL/SQL User's Guide and Reference* for more information.

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 Server Manager, allow you to define variables in your current session. Variables so declared can be used similarly to 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 divide–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.

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 PL/SQL engine of Oracle Server. The PL/SQL capabilities of each Oracle tool are described in the appropriate tool user guide.

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. 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

You can also call a stored procedure from another PL/SQL block, either an anonymous block or another stored procedure. For information on how to call stored procedures from each type of application, see the manual for the specific application tool, such as the *Programmer's Guide to the Oracle Precompilers*.

Dynamic SQL in PL/SQL

You can write stored procedures and anonymous PL/SQL blocks using dynamic SQL. Dynamic SQL statements are not embedded in your source program; rather, they are stored in character strings that are input to, or built by, the program at runtime.

This permits you to create procedures that are more general purpose. For example, using dynamic SQL allows you to create a procedure that operates on a table whose name is not known until runtime.

Additionally, you can parse any data manipulation language (DML) or data definition language (DDL) statement using the DBMS_SQL package. This helps solve the problem of not being able to parse data definition language statements directly using PL/SQL. For example, you might now choose to issue a DROP TABLE statement from within a stored procedure by using the PARSE procedure supplied with the DBMS_SQL package.

CHAPTER

12

Transaction Management

The pigs did not actually work, but directed and supervised the others.

George Orwell: *Animal Farm*

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

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, and 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 something (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 12 – 1 illustrates the banking transaction example.

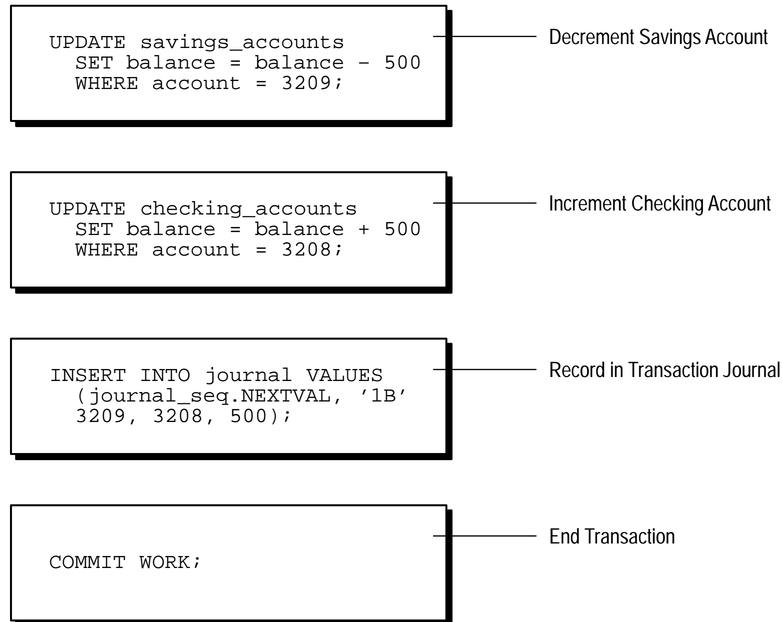


Figure 12 – 1 A Banking Transaction

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 and found to be a valid SQL construction, and that the entire statement executed without error as an atomic unit (for example, all rows of a multi–row 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 said either explicitly or implicitly "make the changes in this transaction permanent". The changes made by the SQL statement(s) of your transaction only become permanent and visible to other users 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 were never executed.

Errors that cause statement rollbacks are errors discovered during the execution stage of SQL statement processing (such as attempting to insert a duplicate value in a primary key or an invalid number into a numeric column), not the parsing stage (such as syntax errors in a SQL statement). Single SQL statements can also be rolled back to resolve *deadlocks* (competition for the same data); see "Deadlock Detection" on page 10–17.

Therefore, 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, the implicit commit that immediately preceded it is not undone. This is a *statement–level rollback.*

> **Note:** Users cannot directly refer to implicit savepoints in rollback statements.

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. See "Transactions and Rollback Segments" on page 3–18 for more information about this topic.

A transaction ends when any of the following occurs:

- You issue a COMMIT or ROLLBACK (without a SAVEPOINT clause) statement.
- 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.

Committing Transactions

Committing a transaction means making permanent the changes performed by the SQL statements within the transaction.

Before a transaction that has modified data is committed, the following will have occurred:

- Oracle has generated rollback segment records in rollback segment buffers of the 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 buffers of the SGA. 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.

After a transaction is committed, the following occurs:

- 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.
- LGWR writes the redo log entries in the redo log buffers of the SGA to the online redo log file. LGWR also writes the transaction's SCN to the online redo log file. This is the atomic event that constitutes the commit of the transaction.
- Oracle releases locks held on rows and tables (see "Locking Mechanisms" on page 10–3 for a discussion of locks).
- Oracle marks the transaction "complete".

> **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 DBWR background process. This action takes place when it is most efficient to do so. As mentioned above, this may happen before the transaction commits or, alternatively, it may happen some time after the transaction commits. See "Oracle Processes" on page 9–5 for more information about the LGWR and DBWR.

Rolling Back Transactions

Rolling back means undoing any changes to data that have been performed by SQL statements within an uncommitted transaction.

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; see the following section, "Savepoints", for a complete explanation of savepoints.

In rolling back **an entire transaction**, without referencing any savepoints, the following occurs:

- Oracle undoes all changes made by all the SQL statements in the transaction by using the corresponding rollback segments.
- Oracle releases all the transaction's locks of data (see "Locking Mechanisms" on page 10–3 for a discussion of locks).
- The transaction ends.

In rolling back a transaction **to a savepoint**, the following occurs:

- Oracle rolls back only the statements executed after the savepoint.
- The specified savepoint is preserved, but all savepoints that were established after the specified one are lost.
- Oracle releases all table and row locks acquired since that savepoint, but retains all data locks acquired previous to the savepoint (see "Locking Mechanisms" on page 10–3 for a discussion of locks).
- The transaction remains active and can be continued.

Savepoints

Intermediate markers or *savepoints* can be declared within the context of a transaction. You use savepoints to divide a long transaction into smaller parts.

Using savepoints, you can arbitrarily mark your work at any point within a long transaction. This allows you the option of later rolling back work performed before the current point in the transaction (the end of 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 also useful in application programs in a similar way. If a procedure contains several functions, 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 then to re–execute the function with revised parameters or perform a recovery action.

After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. Other transactions that were waiting for the previously locked resources can proceed. Other transactions that want to update previously locked rows can do so.

Discrete Transaction Management

Application developers can improve the performance of short, non–distributed transactions by using the procedure BEGIN_DISCRETE_TRANSACTION. This procedure streamlines transaction processing so 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 discreet or not.

Oracle generates redo information, but stores it in a separate location in memory. When the transaction issues a commit request, Oracle writes the redo information to the redo log file (along with other group commits), and applies the changes to the database block directly to the block. Oracle returns control to the application once the commit completes. This eliminates the need to generate undo information, since the block actually is not modified until the transaction is committed, and the redo information is stored in the redo log buffers.

For more information on discrete transactions, see *Oracle7 Server Tuning*.

CHAPTER

13 The Optimizer

I do the very best I know how—the very best I can;
and I mean to keep doing so until the end.

Abraham Lincoln

This chapter discusses how the Oracle optimizer chooses how to execute SQL statements. It includes:

- What Is Optimization?
- How Oracle Optimizes SQL Statements

For more information on the Oracle optimizer, see Chapter 5 of *Oracle7 Server Tuning*.

What Is 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 statement (SELECT, INSERT, UPDATE, or DELETE). There may be many different ways for Oracle to execute such a statement, varying, for example, which tables or indexes are accessed in which order. The procedure used to execute a statement can greatly affect how quickly the statement executes. A part of Oracle called the *optimizer* chooses the way that it believes to be the most efficient.

The optimizer considers a number of factors to make what is usually the best choice among its alternatives. However, an application designer usually knows more about a particular application's data than the optimizer could know. Despite the best efforts of the optimizer, in some situations a developer can choose a more effective way to execute a SQL statement than the optimizer can.

> **Note:** The optimizer may not make the same decisions from one version of Oracle to the next. In future versions of Oracle, the optimizer may make different decisions based on better, more sophisticated information available to it.

Execution Plans

To execute a Data Manipulation Language statement, Oracle may have to perform many steps. Each of these steps either physically retrieves rows of data 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*.

Example

This example shows an execution plan for this SQL statement:

```
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);
```

This statement selects the name, job, salary, and department name for all employees whose salaries do not fall into any recommended salary range.

Figure 13 – 1 shows a graphical representation of the execution plan.

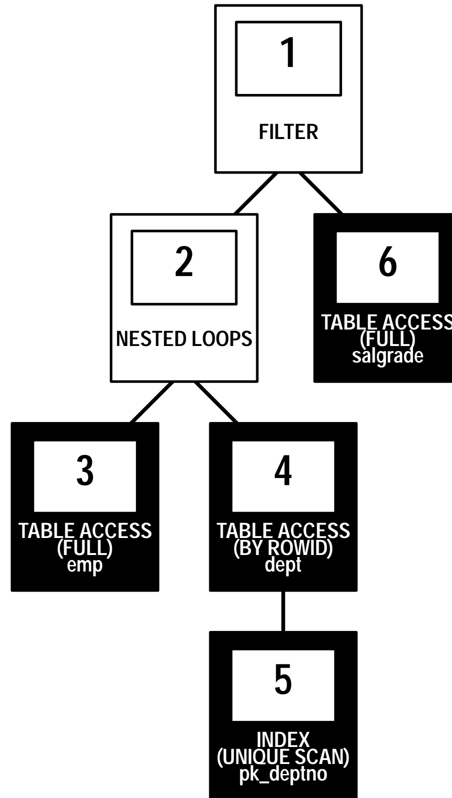


Figure 13 – 1 An Execution Plan

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, returned to the user or application issuing the SQL statement. A set of rows returned by a step is called a *row source.* Figure 13 – 1 is a hierarchical diagram showing the flow of rows from one step to another. The numbering of the steps reflects the order in which they are shown when you view the execution plan, as described in the section "The EXPLAIN PLAN Command", on 13–5. 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 each DEPTNO value returned by step 3 in the PK_DEPTNO index. 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.

Access paths are discussed in the section "Choosing Access Paths" on page 13–11. Methods by which Oracle joins row sources are discussed in *Oracle7 Server Tuning*.

Order of Performing Execution Plan Steps

The steps of the execution plan are not performed in the order in which they are numbered. Oracle first performs the steps that appear as leaf nodes in the tree–structured graphical representation in Figure 13 – 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 13 – 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:
  - Oracle performs Step 5 and returns the resulting ROWID to Step 4.
  - Oracle performs Step 4 and returns the resulting row to Step 2.

– Oracle performs Step 2, accepting a single row from Step 3 and a single row from Step 4, and returning a single row to Step 1.

– Oracle performs Step 6 and returns the resulting row, if any, to Step 1.

– Oracle performs Step 1. If a row is 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. Many parent steps require only a single row from their child steps before they can be executed. For such a parent step, 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 possibly to encompass the rest of the execution plan. 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.

Some parent steps require all rows from their child steps before they can be performed. For such a parent step, Oracle cannot perform the parent step until all rows have been returned from the child step. Such parent steps include sorts, sort–merge joins, group functions, and aggregates.

The EXPLAIN PLAN Command

You can examine the execution plan chosen by the optimizer for a SQL statement by using the EXPLAIN PLAN command. This command causes the optimizer to choose the execution plan and then inserts data describing the plan into a database table. The following 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
```

You can obtain such a listing by using the EXPLAIN PLAN command and then querying the output table. For information on how to use this command and produce and interpret its output, see Appendix A "Performance Diagnostic Tools" of *Oracle7 Server Tuning*.

Each box in Figure 13 – 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 13 – 1.

Oracle's Approaches to Optimization

To choose an execution plan for a SQL statement, the optimizer uses one of these approaches:

rule–based
: Using the *rule–based approach*, the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths in Table 13 – 1 on page 13–13.

cost–based
: Using the *cost–based approach*, the optimizer considers available access paths and factors in information based on statistics in the data dictionary for the objects (tables, clusters, or indexes) accessed by the statement to determine which execution plan is most efficient. The cost–based approach also considers *hints*, or optimization suggestions in the statement placed in a comment.

The Rule–Based Approach

The rule–based approach chooses execution plans based on heuristically ranked operations. 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.

The Cost–Based Approach

Conceptually, the cost–based approach consists of these steps:

1. The optimizer generates a set of potential execution plans for the statement based on its available access paths and hints.
2. The optimizer estimates the cost of each execution plan based on the data distribution and storage characteristics statistics for the tables, clusters, and indexes in the data dictionary.

The *cost* is an estimated value proportional to the expected elapsed time needed to execute the statement using the execution plan. The optimizer calculates the cost based on the estimated computer resources including but not limited to I/O, CPU time, and memory required to execute the statement using the plan. Execution plans with greater costs take more time to execute than those with smaller costs.

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 elapsed time necessary to process all rows accessed by the statement.

Oracle can also optimize a statement with the goal of best *response time*, or minimal elapsed time necessary to process the first row accessed by a SQL statement. For information on how the optimizer chooses an optimization approach and goal, see *Oracle7 Server Tuning*.

Statistics Used for the Cost–Based Approach The cost–based approach uses statistics to estimate the cost of each execution plan. These statistics quantify the data distribution and storage characteristics of tables, columns, and indexes. These statistics are generated using the ANALYZE command. Using these statistics, the optimizer estimates how much I/O, CPU time, and memory are required to execute a SQL statement using a particular execution plan.

The statistics are visible through these data dictionary views:

- 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

For information on these statistics, see the *Oracle7 Server Reference*.

Histograms

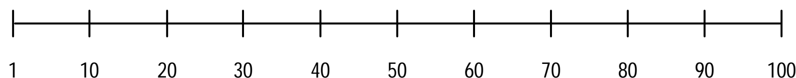
Oracle's cost based optimizer (CBO) uses data value *histograms* to get accurate estimates of the distribution of column data. Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with non–uniform data distributions. You generate histograms by using the ANALYZE command.

One of the fundamental capabilities of any cost–based optimizer is determining the selectivity of predicates that appear in queries. In releases earlier than 7.3, Oracle's cost–based optimizer supported accurate selectivity estimates, assuming that the attribute domains (a table's columns) were uniformly distributed. However, most attribute domains are *not* uniformly distributed.

Histograms enable the cost–based optimizer to describe the distributions of non–uniform domains by using height balanced histograms on specified attributes. Selectivity estimates are used to decide when to use an index and to choose the order that tables are joined.

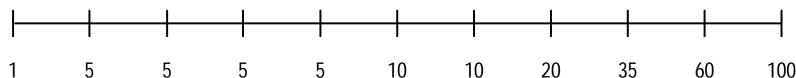
Example

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.

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 the uniform example, 4/10 of the rows had values between 60 and 100, in the non–uniform example, only 1/10 of the rows have values between 60 and 100.

When to Use Histograms

For many users, it is appropriate to use the FOR ALL INDEXED COLUMNS option for creating histograms because indexed columns are typically the columns most often used in WHERE clauses.

You can view histograms by using the following views:

- USER_HISTOGRAMS
- ALL_HISTOGRAMS
- DBA_HISTOGRAMS
- TAB_COLUMNS

Histograms are useful only when they reflect the current data distribution of a given column. If the data distribution is not static, the histogram should be updated frequently. The data need not be static as long as the distribution remains constant.

Histograms can affect performance and should be used only when they substantially improve query plans. 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.

For More Information

See *Oracle7 Server Tuning*.

How Oracle Optimizes SQL Statements

This section explains how Oracle optimizes SQL statements. For any SQL statement processed by Oracle, the optimizer does the following:

evaluation of expressions and conditions
: The optimizer first evaluates expressions and conditions containing constants as fully as possible.

statement transformation
: For a complex statement involving, for example, correlated subqueries, the optimizer may transform the original statement into an equivalent join statement.

view merging
: For a SQL statement that accesses a view, the optimizer often merges the query in the statement with that in the view and then optimizes the result.

choice of optimization approaches	The optimizer chooses either a rule–based or cost–based approach to optimization.
choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain the table's data.
choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, on so on.
choice of join operations	For any join statement, the optimizer chooses an operation to use to perform the join.

Types of SQL Statements

Oracle optimizes these different types of SQL statements:

simple statements	A *simple statement* is an INSERT, UPDATE, DELETE, or SELECT statement that only involves a single table.
simple queries	A *query* is another name for a SELECT statement.
joins	A *join* is a query that selects data from more than one table. A join is characterized by multiple tables in the FROM clause. Oracle pairs the rows from these tables using the condition specified in the WHERE clause and returns the resulting rows. This condition is called the *join condition* and usually compares columns of all the joined tables.
equijoins	An *equijoin* is characterized by a join condition containing an equality operator.
nonequijoins	A *nonequijoin* is characterized by a join condition containing something other than an equality operator.
outer joins	An *outer join* is characterized by a join condition that uses the outer join operator (+) with one or more of the columns of one of the tables. Oracle returns all rows that meet the join condition. Oracle also returns all rows from the table without the outer join operator for which there are no matching rows in the table with the outer join operator.

cartesian products	A join with no join condition results in a cartesian product, or a cross product. A cartesian product is the set of all possible combinations of rows drawn one from each table. In other words, for a join of two tables, each row in one table is matched in turn with every row in the other. A cartesian product for more than two tables is the result of pairing each row of one table with every row of the cartesian product of the remaining tables. All other kinds of joins are subsets of cartesian products effectively created by deriving the cartesian product and then excluding rows that fail the join condition.
complex statements	A complex statement is an INSERT, UPDATE, DELETE, or SELECT statement that contains a form of the SELECT statement called a *subquery*. This is a query within another statement that produces a set of values for further processing within the statement. The outer portion of the complex statement that contains a subquery is called the *parent statement*.
compound queries	A *compound query* is a query that uses set operators (UNION, UNION ALL, INTERSECT, or MINUS) to combine two or more simple or complex statements. Each simple or complex statement in a compound query is called a *component query*.
statements accessing views	You can also write simple, join, complex, and compound statements that access views as well as tables.
distributed statements	A *distributed statement* is one that accesses data on a remote database.

Choosing Access Paths

One of the most important choices the optimizer makes when formulating an execution plan is how to retrieve the data from the database. For any row in any table accessed by a SQL statement, there may be many access paths by which that row can be located and retrieved. The optimizer chooses one of them.

This section discusses these topics:

- the basic methods by which Oracle can access data
- each access path and when it is available to the optimizer
- how the optimizer chooses among available access paths

Access Methods

This section describes basic methods by which Oracle can access data.

Full Table Scans A full table scan retrieves rows from a table. To perform a full table scan, Oracle reads all rows in the table, examining each row to determine whether it satisfies the statement's WHERE clause. Oracle reads every data block allocated to the table sequentially, so a full table scan can be performed very efficiently using multiblock reads. Oracle reads each data block only once.

Table Access by ROWID A table access by ROWID also retrieves rows from a table. The ROWID of a row specifies the datafile and data block containing the row and the location of the row in that block. Locating a row by its ROWID is the fastest way for Oracle to find a single row.

To access a table by ROWID, Oracle first obtains the ROWIDs of the selected rows, either from the statement's WHERE clause or through an index scan of one of more of the table's indexes. Oracle then locates each selected row in the table based on its ROWID.

Cluster Scans From a table stored in an indexed cluster, a cluster scan retrieves rows that have the same cluster key value. In an indexed cluster, all rows with the same cluster key value are stored in the same data blocks. To perform a cluster scan, Oracle first obtains the ROWID of one of the selected rows by scanning the cluster index. Oracle then locates the rows based on this ROWID.

Hash Scans Oracle can use a hash scan to locate rows in a hash cluster based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data blocks. To perform a hash scan, Oracle first obtains the hash value by applying a hash function to a cluster key value specified by the statement. Oracle then scans the data blocks containing rows with that hash value.

Index Scans An index scan retrieves data from an index based on the value of one or more columns of the index. To perform an index scan, Oracle searches the index for the indexed column values accessed by the statement. If the statement accesses only columns of the index, Oracle reads the indexed column values directly from the index, rather than from the table.

In addition to each indexed value, an index also contains the ROWIDs of rows in the table having that value. If the statement accesses other columns in addition to the indexed columns, Oracle then finds the rows in the table with a table access by ROWID or a cluster scan.

An index scan can be one of these types:

Unique
: A unique scan of an index returns only a single ROWID. Oracle can only perform a unique scan in cases in which only a single ROWID, rather than many ROWIDs, is required. For example, Oracle performs a unique scan if there is a UNIQUE or a PRIMARY KEY constraint that guarantees that the statement accesses only a single row.

Range
: A range scan of an index can return one or more ROWIDs depending on how many rows are accessed by the statement.

Access Paths

Table 13 – 1 lists the access paths. The rank of each path is used by the rule–based approach to choose a path when more than one path is available.

Rank	Access Path
1	Single row by ROWID
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single–column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort–merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed columns
15	Full table scan

Table 13 – 1 Access Paths

The optimizer can only choose to use a particular access path for a table if the statement contains a WHERE clause condition or other construct that makes that access path available. Each of the following sections describes an access path and discusses

- when it is available
- the method Oracle uses to access data with it
- the output generated for it by the EXPLAIN PLAN command

Choosing Among Access Paths

This section describes how the optimizer chooses among available access paths:

- when using the rule–based approach
- when using the cost–based approach

Choosing Among Access Paths with the Rule–Based Approach With the rule–based approach, the optimizer chooses whether to use an access path based on these factors:

- the available access paths for the statement
- the ranks of these access paths in Table 13 – 1

To choose an access path, the optimizer first examines the conditions in the statement's WHERE clause to determine which access paths are available. The optimizer then chooses the most highly ranked available access path. Note that the full table scan is the lowest ranked access path on the list. This means that the rule–based approach always chooses an access path that uses an index if one is available, even if a full table scan might execute faster.

The order of the conditions in the WHERE clause does not normally affect the optimizer's choice among access paths.

Example Consider this SQL statement that selects the employee numbers of all employees in the EMP table with an ENAME value of 'CHUNG' and with a SAL value greater than 2000:

```
SELECT empno
    FROM emp
    WHERE ename = 'CHUNG'
      AND sal > 2000;
```

Consider also that the EMP table has these integrity constraints and indexes:

- There is a PRIMARY KEY constraint on the EMPNO column that is enforced by the index PK_EMPNO.
- There is an index named ENAME_IND on the ENAME column.
- There is an index named SAL_IND on the SAL column.

Based on the conditions in the WHERE clause of the SQL statement, the integrity constraints, and the indexes, these access paths are available:

- A single–column index access path using the ENAME_IND index is made available by the condition ENAME = 'CHUNG'. This access path has rank 9.
- An unbounded range scan using the SAL_IND index is made available by the condition SAL> 2000. This access path has rank 11.
- A full table scan is automatically available for all SQL statements. This access path has rank 15.

Note that the PK_EMPNO index does not make the single row by primary key access path available because the indexed column does not appear in a condition in the WHERE clause.

Using the rule–based approach, the optimizer chooses the access path that uses the ENAME_IND index to execute this statement. The optimizer chooses this path because it is the most highly ranked path available.

Choosing Among Access Paths with the Cost–Based Approach With the cost–based approach, the optimizer chooses an access path based on these factors:

- the available access paths for the statement
- the estimated cost of executing the statement using each access path or combination of paths

To choose an access path, the optimizer first determines which access paths are available by examining the conditions in the statement's WHERE clause. The optimizer then generates a set of possible execution plans using available access paths and estimates the cost of each plan using the statistics for the index, columns, and tables accessible to the statement. The optimizer then chooses the execution plan with the lowest estimated cost.

The optimizer's choice among available access paths can be overridden with hints. For information on hints, see *Oracle7 Server Tuning*.

To choose among available access paths, the optimizer considers these factors:

- **Selectivity:** The *selectivity* is the percentage of rows in the table that the query selects. Queries that select a small percentage of a table's rows have good selectivity, while a query that selects a large percentage of the rows has poor selectivity.

  The optimizer is more likely to choose an index scan over a full table scan for a query with good selectivity than for one with poor selectivity. Index scans are usually more efficient than full table scans for queries that access only a small percentage of a table's rows, while full table scans are usually faster for queries that access a large percentage.

  To determine the selectivity of a query, the optimizer considers these sources of information:

  - the operators used in the WHERE clause
  - unique and primary key columns used in the WHERE clause
  - statistics for the table

  The following examples in this section illustrate the way the optimizer uses selectivity.

- **DB_FILE_MULTIBLOCK_READ_COUNT**: Full table scans use multiblock reads, so the cost of a full table scan depends on the number of multiblock reads required to read the entire table, which depends on the number of blocks read by a single multiblock read, which is specified by the initialization parameter DB_FILE_MULTIBLOCK_READ_COUNT. For this reason, the optimizer may be more likely to choose a full table scan when the value of this parameter is high.

Example

Consider this query that uses an equality condition in its WHERE clause to select all employees name Jackson:

```
SELECT *
    FROM emp
    WHERE ename = 'JACKSON';
```

If the ENAME column is a unique or primary key, the optimizer determines that there is only one employee named Jackson, and the query returns only one row. In this case, the query is very selective, and

the optimizer is most likely to access the table using a unique scan on the index that enforces the unique or primary key (access path 4).

Example Consider again the query in the previous example. If the ENAME column is not a unique or primary key, the optimizer can use these statistics to estimate the query's selectivity:

USER_TAB_COLUMNS.NUM_DISTINCT	This statistic is the number of values for each column in the table.
USER_TABLES.NUM_ROWS	This statistic is the number of rows in each table.

By dividing the number of rows in the EMP table by the number of distinct values in the ENAME column, the optimizer estimates what percentage of employees have the same name. By assuming that the ENAME values are uniformly distributed, the optimizer uses this percentage as the estimated selectivity of the query.

Example Consider this query that selects all employees with employee ID numbers less than 7500:

```
SELECT *
    FROM emp
    WHERE empno < 7500;
```

To estimate the selectivity of the query, the optimizer uses the boundary value of 7500 in the WHERE clause condition and the values of the HIGH_VALUE and LOW_VALUE statistics for the EMPNO column if available. These statistics can be found in the USER_TAB_COLUMNS view. The optimizer assumes that EMPNO values are evenly distributed in the range between the lowest value and highest value. The optimizer then determines what percentage of this range is less than the value 7500 and uses this value as the estimated selectivity of the query.

Optimizing Distributed Statements

The optimizer chooses execution plans for SQL statements that access data on remote databases in much the same way it chooses executions for statements that access only local data:

- If all the tables accessed by a SQL statement are collocated on the same remote database, Oracle sends the SQL statement to that remote database. The remote Oracle instance executes the statement and sends only the results back to the local database.
- If a SQL statement accesses tables that are located on different databases, Oracle decomposes the statement into individual fragments, each of which access tables on a single database.

Oracle then sends each fragment to the database it accesses. The remote Oracle instance for each of these databases executes its fragment and returns the results to the local database.

When choosing the execution plan for a distributed statement, the optimizer considers the available indexes on remote databases just as it does indexes on the local database. If the statement uses the cost–based approach, the optimizer also considers statistics on remote databases. Furthermore, the optimizer considers the location of data when estimating the cost of accessing it. For example, a full scan of a remote table has a greater estimated cost than a full scan of an identical local table.

PART

VI

Programmatic Constructs

CHAPTER

14 Procedures and Packages

We're dealing here with science, but it is science which has not yet been fully codified by scientific minds. What we have are the memoirs of poets and occult adventurers...

Anne Rice: *The Tale of the Body Thief*

This chapter discusses the procedural capabilities of Oracle. It includes:

- An Introduction to Stored Procedures and Packages
- Procedures and Functions
- Packages
- How Oracle Stores Procedures and Packages
- How Oracle Executes Procedures and Packages

The information in this chapter applies only to those systems using Oracle with the procedural option. For information about the dependencies of procedures, functions, and packages, and how Oracle manages these dependencies, see Chapter 16, "Dependencies Among Schema Objects".

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide.*

An 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.

Stored Procedures and Functions

A procedure or function is a schema object that logically groups 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 using an Oracle tool, such as SQL*Plus, or call it explicitly in the code of a database application, such as an Oracle Forms or Precompiler application, or in the code of another procedure or trigger. Figure 14 – 1 illustrates a simple procedure stored in the database, being called by several different database applications.

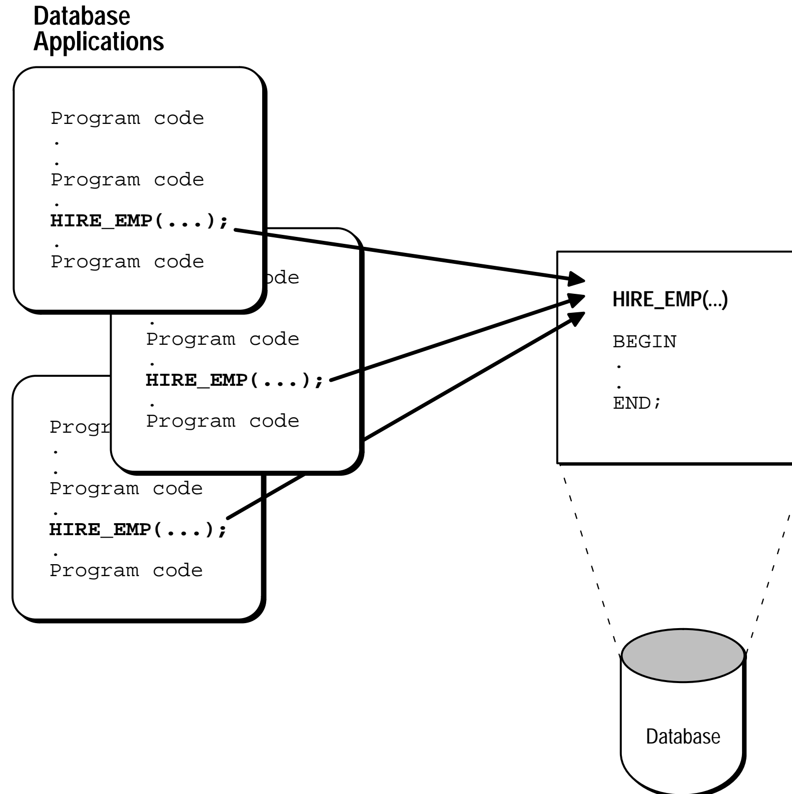


Figure 14 – 1 A Stored Procedure

The stored procedure in Figure 14 – 1, which inserts an employee record into the EMP table, is shown in Figure 14 – 2.

```
Procedure HIRE_EMP (name VARCHAR2, job VARCHAR2,
   mgr NUMBER, hiredate DATE, sal NUMBER,
   comm NUMBER, deptno NUMBER



BEGIN
.
.
INSERT INTO emp VALUES
   (emp_sequence.NEXTVAL, name, job, mgr
   hiredate, sal, comm, deptno);
.
.
END;
```

Figure 14 – 2 The HIRE_EMP Procedure

All of the database applications in Figure 14 – 1 call the HIRE_EMP procedure. Alternatively, a privileged user might use Server Manager 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.

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 14 – 3 illustrates a package that encapsulates a number of procedures used to manage an employee database.

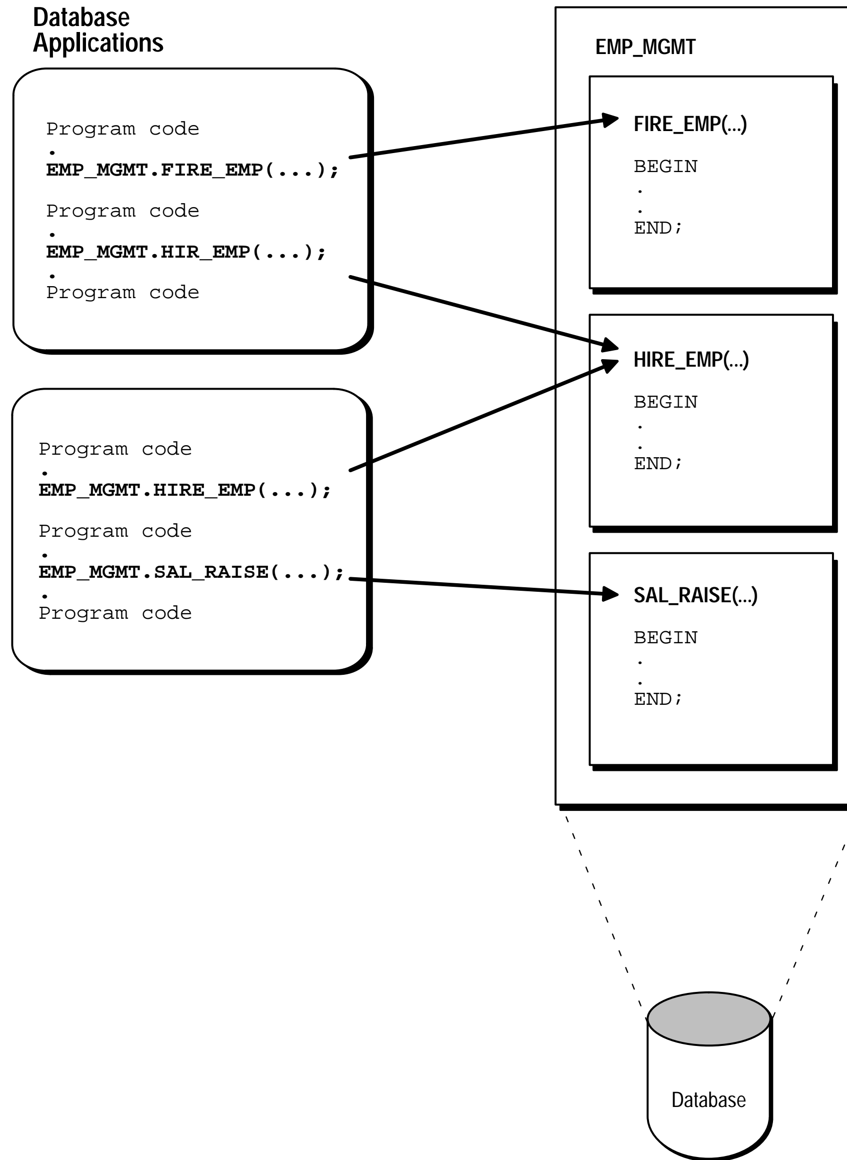


Figure 14 – 3 A Stored Package

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, the following statement might be issued using Server Manager 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. These advantages are described in the section "Packages" on page 14–9.

PL/SQL

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 in it. 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 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.

For more information about PL/SQL, see the section "PL/SQL" on page 11–7.

Procedures and Functions

Oracle can process procedures and functions as well as individual SQL statements. 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 both SQL and PL/SQL statements are included in the CREDIT_ACCOUNT procedure.

Procedures and functions are nearly identical. The only differences are that functions always return a single value to the caller, while procedures do not. For simplicity, the term "procedure" is used in the remainder of this chapter to mean "procedures and functions," unless otherwise noted.

How Procedures Are Used

You should design and use all stored procedures so that they have the following properties:

- Define procedures to complete a single, focused task. Do not define long procedures with several distinct subtasks, or 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.

Applications for Procedures

Procedures provide advantages in the following areas:

- security
- performance
- memory allocation
- productivity
- integrity

Security

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.

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 that have only the privilege to execute the procedure and 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.

Performance

Stored procedures can improve database performance. Use of procedures dramatically reduces the amount of information that must be sent over a network 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. Furthermore, because a procedure's compiled form is readily available in the database, no compilation is required at execution time. Additionally, 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 rows 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.

Anonymous PL/SQL Blocks vs. Stored Procedures

You create an anonymous PL/SQL block by sending an unnamed PL/SQL block to 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 subsequent reuse.

Shared SQL allows a compiled anonymous PL/SQL block already in the shared pool to be reused and shared until it is flushed out of the shared pool.

Alternatively, a stored procedure is created and stored in the database as an 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.

In summary, by moving PL/SQL blocks out of a database application and into stored database procedures, you avoid unnecessary procedure recompilations by Oracle at runtime, improving the overall performance of the application and Oracle.

Standalone Procedures vs. Package 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 "Packages" on page 14–9 for information on the advantages of packages.

Dependency Tracking for Stored Procedures

A stored procedure is dependent 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 Chapter 16, "Dependencies Among Schema Objects", for more information about dependency tracking.

Packages

Packages provide a method of encapsulating related procedures, functions, and associated cursors and variables together as a unit in the database.

For example, the following two statements create the specification and body for a package that contains several procedures and functions that process banking transactions.

```
CREATE PACKAGE bank_transactions  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;
```

(continued next page)

```
  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;
```

(continued next page)

```
  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;
```

(continued next page)

```
  PROCEDURE enter_transaction (acct   NUMBER,
                   kind   CHAR,
                   amount NUMBER) IS

/* Enters a bank transaction into the TRANSACTIONS table. A new
   transaction is always input 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;
```

While packages allow the database administrator or application developer to organize similar routines, they also offer increased functionality and database performance.

Applications for 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
- separation of the package specification and package body
- better performance

Encapsulation

Stored packages allow you to encapsulate, or group, related stored procedures, variables, datatypes, etc. in a single named, stored unit in the database. This provides for 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. However, the package can be defined 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, which are described in Chapter 17, "Database Access".

Separate Package Specification and Package Body

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:

- By defining the package specification separately from the package body, 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. For more information about dependencies, see Chapter 16, "Dependencies Among Schema Objects".

Performance Improvement

Using packages rather than stand–alone stored procedures results in the following improvements:

- The 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, objects that reference a package's constructs (always via the specification) never need to 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 is dependent on the objects referenced by the procedures and functions defined in its body. Oracle automatically tracks and manages such dependencies. See Chapter 16, "Dependencies Among Schema Objects", for more information about dependency tracking.

How Oracle Stores Procedures and Packages

When you create a procedure or package, Oracle automatically performs these steps:

1. Compiles the procedure or package.
2. Stores the compiled code in memory.
3. 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. Information on identifying compilation errors is contained in the *Oracle7 Server Application Developer's Guide.*

Storing the Compiled Code in Memory

Oracle caches the compiled procedure or package in the shared pool of the 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/her session. See "The Shared Pool" on page 9–20 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 a procedure or package in the database:

object name	Oracle uses this name to identify 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. The database administrator should plan the size of the SYSTEM tablespace, keeping 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 these steps to execute it:

1. Verifies user access.
2. Verifies procedure validity.
3. Executes the procedure.

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 see if 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 objects referenced within the procedure or package (such as tables, views, and other procedures) have been altered or dropped (for example, if 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 object privilege for one or more of the 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, the compiled code is executed.

If an invalid standalone or packaged procedure is called, it is automatically recompiled before being executed.

For a complete discussion of valid and invalid procedures and packages, recompiling procedures, and a thorough discussion of dependency issues, see Chapter 16, "Dependencies Among Schema Objects".

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 11 – 1 on page 11–8.

CHAPTER

15 Database Triggers

You may fire when you are ready, Gridley.

George Dewey: *at the battle of Manila Bay*

This chapter discusses database triggers; that is, procedures that are stored in the database and implicitly executed ("fired") when a table is modified. This chapter includes:

- An Introduction to Triggers
- Parts of a Trigger
- Triggers Execution

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide.*

An Introduction to Triggers

Oracle allows you to define procedures that are implicitly executed when an INSERT, UPDATE, or DELETE statement is issued against the associated table. These procedures are called database triggers.

Triggers are similar to stored procedures, discussed in Chapter 14, "Procedures and Packages". A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. While a procedure is explicitly executed by a user, application, or trigger, one or more triggers are implicitly fired (executed) by Oracle when a triggering INSERT, UPDATE, or DELETE statement is issued, no matter which user is connected or which application is being used.

For example, Figure 15 – 1 shows a database application with some SQL statements that implicitly fire several triggers stored in the database.

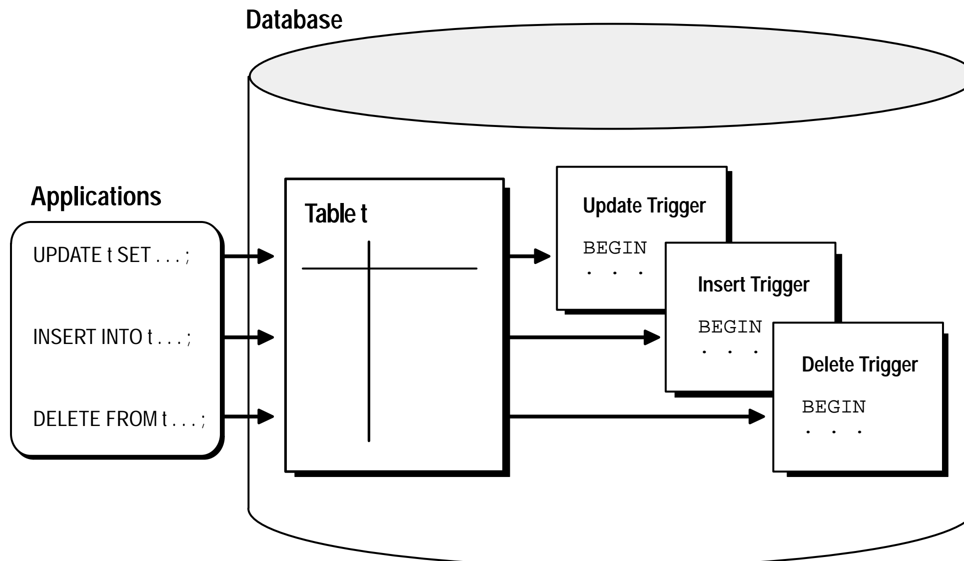


Figure 15 – 1 Triggers

Notice that triggers are stored in the database separately from their associated tables.

Triggers can be defined only on tables, not on views. However, triggers on the base table(s) of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against a view.

How Triggers Are Used

In many cases, triggers supplement the standard capabilities of Oracle to provide a highly customized database management system. For example, a trigger can permit DML operations against a table only if they are issued during regular business hours. The standard security features of Oracle, roles and privileges, govern which users can submit DML statements against the table. In addition, the trigger further restricts DML operations to occur only at certain times during weekdays. This is just one way that you can use triggers to customize information management in an Oracle database.

In addition, triggers are commonly used 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

Examples of many of these different trigger uses are included in the *Oracle7 Server Application Developer's Guide.*

A Cautionary Note about Trigger Use

When a trigger is fired, a SQL statement within its trigger action potentially can fire other triggers, as illustrated in Figure 15 – 2. When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading.*

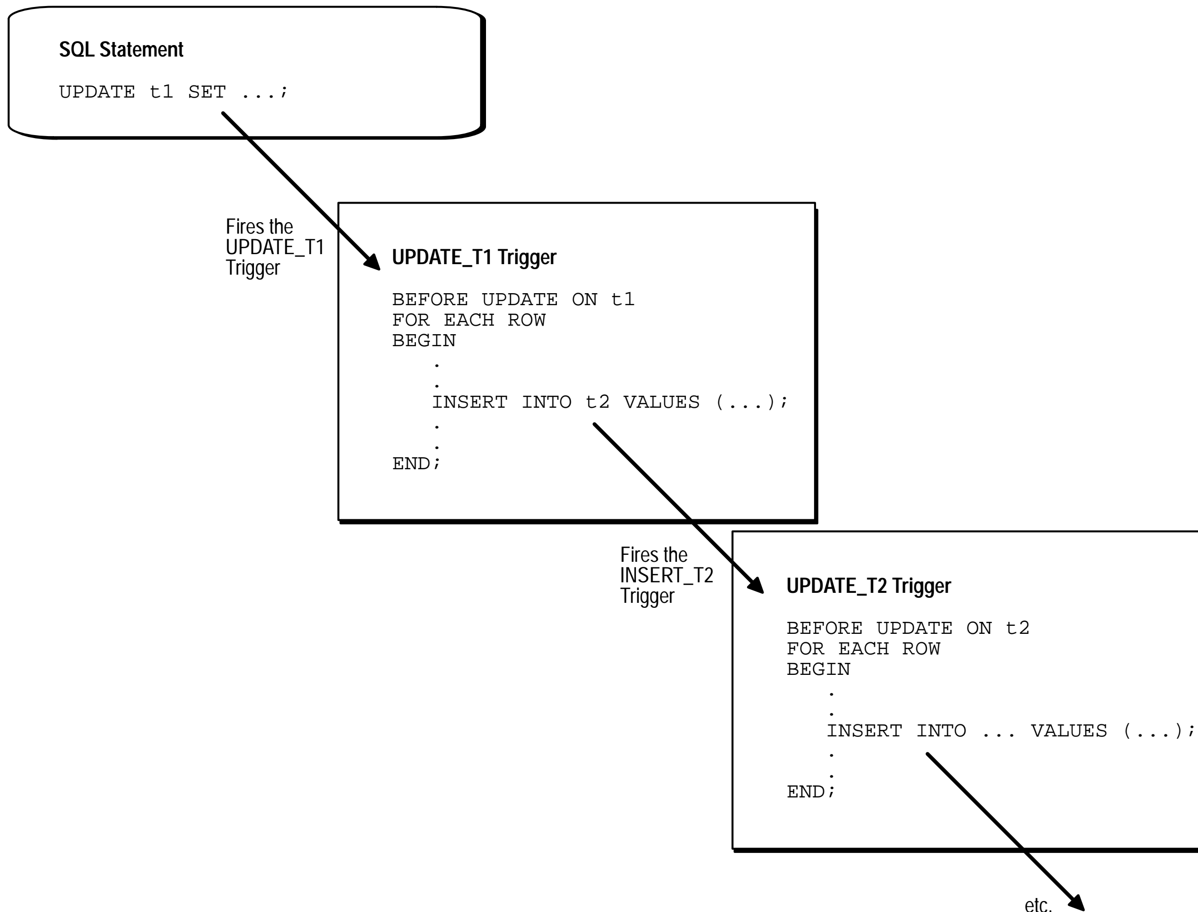


Figure 15 – 2 Cascading Triggers

While triggers are useful for customizing a database, you should only use triggers when necessary. The excessive use of triggers can result in complex interdependences, which may be difficult to maintain in a large application.

Database Triggers vs. Oracle Forms Triggers

Oracle Forms can also define, store, and execute triggers. However, do not confuse Oracle Forms triggers with the database triggers discussed in this chapter.

Database triggers are defined on a table, stored in the associated database, and executed as a result of an INSERT, UPDATE, or DELETE statement being issued against a table, no matter which user or application issues the statement.

Oracle Forms triggers are part of an Oracle Forms application and are fired only when a specific trigger point is executed within a specific Oracle Forms application. SQL statements within an Oracle Forms application, as with any database application, can implicitly cause the firing of any associated database trigger. For more information about Oracle Forms and Oracle Forms triggers, see the *Oracle Forms User's Guide*.

Triggers vs. Declarative Integrity Constraints

Triggers and declarative integrity constraints can both be used to constrain data input. However, triggers and integrity constraints have significant differences.

A declarative integrity constraint is a statement about the database that is never false while the constraint is enabled. A constraint applies to existing data in the table and any statement that manipulates the table.

Triggers constrain what transactions can do. A trigger does not apply to data loaded before the definition of the trigger. Therefore, it does not guarantee all data in a table conforms to its rules.

A trigger enforces transitional constraints; that is, a trigger only enforces a constraint at the time that the data changes. Therefore, a constraint such as "make sure that the delivery date is at least seven days from today" should be enforced by a trigger, not a declarative integrity constraint.

In evaluating triggers that contain SQL functions that have NLS parameters as arguments (for example, TO_CHAR, TO_DATE, and TO_NUMBER), the default values for these parameters are taken from the NLS parameters currently in effect for the session. You can override the default values by specifying NLS parameters explicitly in such functions when you create a trigger.

For more information about declarative integrity constraints, see Chapter 7, "Data Integrity".

Parts of a Trigger

A trigger has three basic parts:

- a triggering event or statement
- a trigger restriction
- a trigger action

Figure 15 – 3 represents each of these parts of a trigger and is not meant to show exact syntax. Each part of a trigger is explained in greater detail in the following sections.

REORDER Trigger

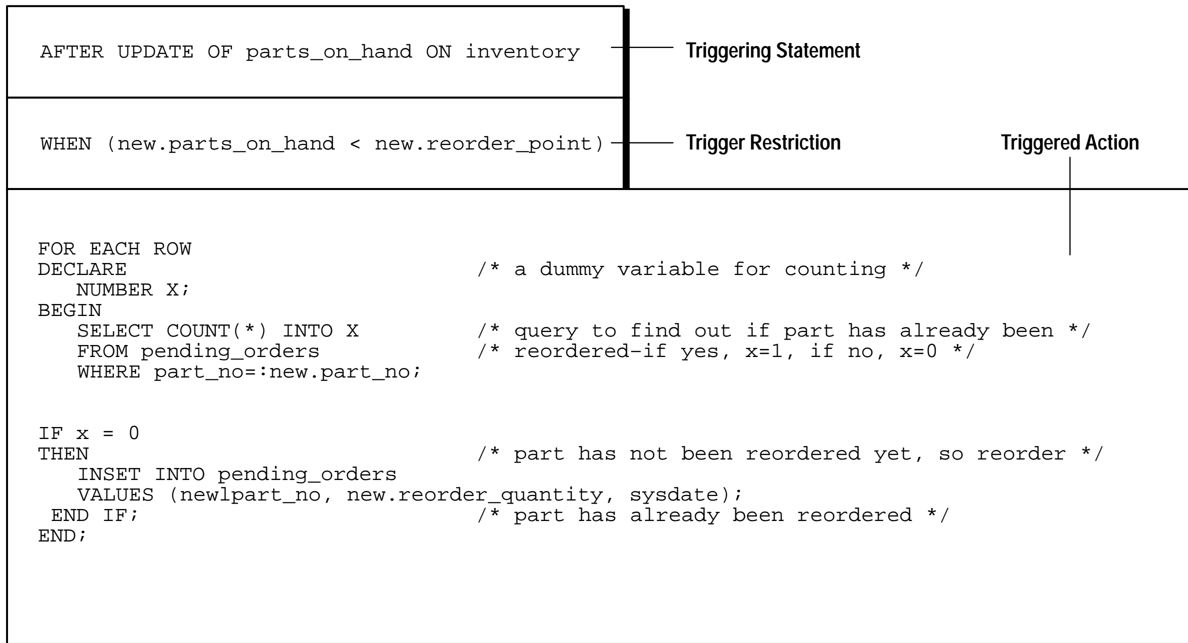


Figure 15 – 3 The REORDER Trigger

Triggering Event or Statement

A triggering event or statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement on a table.

For example, in Figure 15 – 3, the triggering statement is

```
. . . UPDATE OF parts_on_hand ON inventory . . .
```

which means that when the PARTS_ON_HAND column of a row in the INVENTORY table is updated, fire the trigger. Note that 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. Because INSERT and DELETE statements affect entire rows of information, a column list cannot be specified for these options.

A triggering event can specify multiple DML statements, as in

```
. . . INSERT OR UPDATE OR DELETE OF inventory . . .
```

which means that when an INSERT, UPDATE, or DELETE statement is issued against the INVENTORY table, fire the trigger. When multiple types of DML statements can fire a trigger, conditional predicates can be used to detect the type of triggering statement. Therefore, a single trigger can be created that executes different code based on the type of statement that fired the trigger.

Trigger Restriction

A trigger restriction specifies a Boolean (logical) 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.

A trigger restriction is an option available for triggers that are fired for each row. Its function is to control the execution of a trigger conditionally. You specify a trigger restriction using a WHEN clause. For example, the REORDER trigger in Figure 15 – 3 has a trigger restriction. The trigger is fired by an UPDATE statement affecting the PARTS_ON_HAND column of the INVENTORY table, but the trigger action only fires if the following expression is TRUE:

```
new.parts_on_hand < new.reorder_point
```

Trigger Action

A trigger action is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE.

Similar to stored procedures, a trigger action can contain SQL and PL/SQL statements, define PL/SQL language constructs (variables, constants, cursors, exceptions, and so on), and call stored procedures. Additionally, for row trigger, the statements in a trigger action have access to column values (new and old) of the current row being processed by the trigger. Two correlation names provide access to the old and new values for each column.

Types of 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 might be fired by an UPDATE statement that updates many rows), or 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 at all.

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 15 – 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, regardless of how many rows are deleted from the table.

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, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

BEFORE vs. AFTER Triggers

When defining a trigger, you can specify the *trigger timing*. That is, you can specify 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 Triggers BEFORE triggers execute the trigger action before the triggering statement. This type of trigger is commonly used in the following situations:

- BEFORE triggers are used when the trigger action should determine whether the triggering statement should be allowed to complete. By 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.
- BEFORE triggers are used 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. AFTER triggers are used in the following situations:

- AFTER triggers are used when you want the triggering statement to complete before executing the trigger action.
- If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

Combinations

Using the options listed in the previous two sections, you can create four types of 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 statement trigger**
  After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
- **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.

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. Figure 15 – 4 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. A 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, and finally the statistical information is saved in the table STAT_TAB by the AFTER statement trigger.

```
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;
/
```

Figure 15 – 4 Sample Package and Trigger for SAL Table

Trigger Execution

A trigger can be 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

- 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 objects referenced in the code of the trigger action
- uses two–phase commit if a trigger updates remote tables in a distributed database
- if more than one trigger of the same type for a given statement exists, Oracle fires each of those triggers in an unspecified order

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, and 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 on tables changed by
                   AFTER statement triggers checked.
           6.  Statements of BEFORE row triggers executed.
           7.  Integrity constraint on tables changed by
               BEFORE row triggers checked.
       8.  SQL statement executed.
       9.  Integrity constraint from SQL statement checked.
```

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:** Be aware that triggers of different types are fired in a specific order. However, 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 not to 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 contained 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 before proceeding.

The following examples illustrate these points.

Example

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

Assume the following definition of the TOTAL_SALARY trigger, a trigger to maintain 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 for Triggers

For release 7.3, Oracle stores triggers in their 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 a trigger is flushed from the shared pool.

For More Information

See "How Oracle Stores Procedures and Packages" on page 14–14.

Execution of Triggers

Oracle internally executes a trigger using the same steps used for procedure execution. The subtle and only difference is that a user automatically has the right to fire a trigger if he/she has the privilege to execute the triggering statement. Other than this, triggers are validated and executed the same way as stored procedures.

For More Information

See "How Oracle Executes Procedures and Packages" on page 14–15.

Dependency Maintenance for Triggers

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 dependency issues for stored procedures. In releases earlier than 7.3, triggers were kept in memory. In release 7.3, triggers are treated like stored procedures; they are inserted in the data dictionary. Like procedures, triggers are dependent on referenced objects. Oracle automatically manages dependencies among objects.

For More Information

See Chapter 16, "Dependencies Among Schema Objects".

CHAPTER

16 Dependencies Among Schema Objects

Whoever you are — I have always depended on the kindness of strangers.

Tennessee Williams: *A Streetcar Named Desire*

The definitions of certain objects, such as views and procedures, reference other objects, such as tables. Therefore, some objects are dependent on the objects referenced in their definitions. This chapter discusses the dependencies among objects and how Oracle automatically tracks and manages such dependencies. It includes:

- An Introduction to Dependency Issues
- Advanced Dependency Management Topics

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for more information on schema object dependencies in that environment.

An 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 16 – 1 illustrates the different types of dependent and referenced objects.

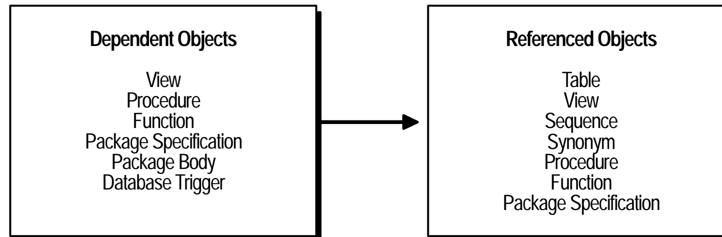


Figure 16 – 1 Types of Possible Dependent and Referenced Schema Objects

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 can be used.

Oracle automatically records dependencies among objects to alleviate the complex job of dependency management from 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 objects, all schema objects in a database have a status:

INVALID — The object must be compiled before it can be used. In the case of procedures, functions, and packages, this means compiling the 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 there were some errors relating to the object. 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. Objects with such problems remain invalid.
VALID	The object has been compiled and can be immediately used when referenced.

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 table(s) of that view. Therefore, if you alter a base table, the view is invalidated, which then invalidates the stored procedure. Figure 16 – 2 illustrates this.

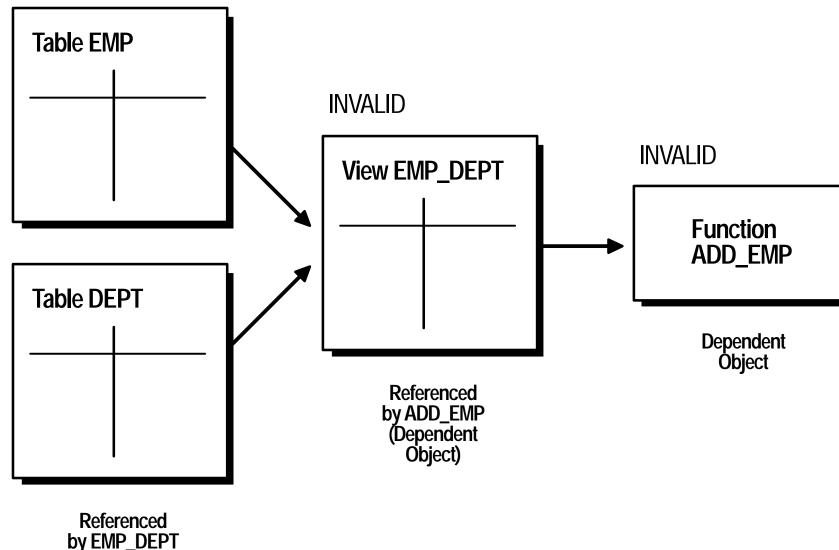


Figure 16 – 2 Indirect Dependencies

When an object is referenced (directly in a SQL statement or indirectly via 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. Depending on the status of the objects that are directly and indirectly referenced in a SQL statement, different events can occur.

- If every referenced object is valid, the SQL statement executes immediately without any additional work.
- If any referenced view or procedure (including functions and packages) is invalid, Oracle automatically attempts to compile the object. If all referenced objects that are invalid can be successfully compiled, the objects are compiled, and the SQL statement executes successfully. If an invalid object cannot be successfully compiled, the object remains invalid, an error is returned, and the transaction containing the SQL statement is rolled back.

  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 true:

- The definition of the view or program unit is correct; all SQL and PL/SQL statements must be proper constructs.
- All referenced objects are 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 has 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, 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, 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.

Attempting 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:

- All base tables referenced by the defining query of a view must exist. Therefore, 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, 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, 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 . . ." 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, the dependent procedure is invalidated.
- If the base table of the referenced view is altered, the view and the dependent procedure are invalidated.
- If the referenced standalone procedure is replaced, the dependent procedure is invalidated.
- If the body of the referenced package is replaced, the dependent procedure is not affected. However, if the specification of the referenced package is replaced, the dependent procedure is invalidated.

This last case reveals a mechanism for minimizing dependencies among procedures and referenced objects.

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.

Advanced Dependency Management Topics

The previous section described conceptually the dependency management mechanisms of Oracle. The following sections offer additional information about Oracle's automatic dependency management features. For information on forcing the recompilation of an invalid view or program unit, see the *Oracle7 Server Application Developer's Guide*. If you are using Trusted Oracle, also see the *Trusted Oracle7 Server Administrator's Guide*.

Dependency Management and Non–Existent Schema Objects

When a dependent object is created, Oracle attempts to resolve all references by first searching in the current schema. 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. If a private synonym is not found, Oracle moves on, looking for a public synonym. If a public synonym is not found, Oracle searches for a schema name that matches the first portion of the object name. If a matching schema name is found, Oracle attempts to find the object in that schema. 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 non–existence of other objects. This 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 PUBLIC.
- 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 non–existence 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;
```

> **Note:** 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 "non–existent" objects — objects that, if they existed, would change the interpretation of the object's definition. Such dependencies must be noted in case a non–existent object is later created. If a non–existent object is created, all dependent objects must be invalidated so that dependent objects can be recompiled and verified.

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, dependencies on non–existent objects checked during object resolution must be managed in case the non–existent object is later created.

Shared SQL Dependency Management

In addition to managing the dependencies among schema objects, Oracle also manages the 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.

Package Invalidations and Session State

Each session that references a package construct has its own instance of the corresponding 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 (specification or body) are subsequently invalidated and recompiled.

Local and Remote Dependency Management

Tracking dependencies and completing necessary recompilations are important tasks automatically performed by Oracle. In the simplest case, dependencies must be managed among the objects in a single database (local dependency management). For example, a statement in a procedure can reference a table in the same database. In more complex systems, Oracle must manage the dependencies in distributed environments across a network (remote dependency management). 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 dependency checking using the database's internal "depends–on" table, which keeps track of each 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 that there is a stored procedure UPDATE_SAL that 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. This implies that the procedure cannot be executed until the procedure 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", which incurs less overhead than a full recompilation.

Managing Remote Dependencies

Application–to–database and distributed database dependencies must also be considered. For example, an Oracle Forms application can contain a trigger that references a table, or a local stored procedure can call a remote procedure in a distributed database system. The database system must account for dependencies among such objects. Oracle manages remote dependencies using different mechanisms, 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*. For example, when a procedure is compiled, such as during creation or subsequent replacement, its timestamp (the time it is created, altered, or replaced) is recorded in the data dictionary. Additionally, the compiled version of the procedure includes information (such as schema, package name, procedure name, and timestamp) for each remote procedure it references.

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, previous to an invalid procedure call, might execute successfully, while 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.

Dependencies Among Other Remote Schema Objects Dependencies among remote schema objects other than local procedure–to–remote procedure dependencies are not managed by Oracle.

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). Lack of dependency management is preferable in such cases to avoid 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, and 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

VII

Database Security

CHAPTER

17

Database Access

Allow me to congratulate you, sir. You have the most totally closed mind that I've ever encountered!

Jon Pertwee (as the Doctor): *Frontier in Space*

Database security involves allowing or disallowing users from performing actions on the database and the objects within it. 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. Because privileges are granted to users at the discretion of other users, this is called discretionary security.

This chapter explains how access to Oracle is controlled. It includes:

- Schemas, Database Users, and Security Domains
- User Authentication
- User Tablespace Settings and Quotas
- The User Group PUBLIC
- User Resource Limits and Profiles
- Licensing

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for information on database access in that environment.

Schemas, Database Users, and Security Domains

Schemas and users help database administrators manage database security. A *schema* is a named collection of schema objects, such as tables, views, clusters, procedures, and packages. A *user* (sometimes called a *username*) is a name defined in the database that can connect to and access objects in database schemas.

To access a database, a user must run a database application (such as an Oracle Forms form, SQL*Plus, or a Precompiler program) and connect using a username defined in the database.

When a database user is created, a corresponding schema of the same name is created for the user. By default, once a user connects to a database, the user has access to all objects contained in the corresponding schema. A user is associated only with the schema of the same name; therefore, the terms user and schema are similar.

The access rights of a user are controlled by the different settings of the user's security domain. 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, if any, of tablespaces accessible to the user and the associated quotas for each listed tablespace
- the user's resource limit profile; that is, limits that dictate the amount of system resources available to the user
- the privileges and roles that provide the user with appropriate access to objects needed to perform database operations

This chapter describes the security domain options listed above, except for privileges and roles, which are discussed in Chapter 18, "Privileges and Roles".

> **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. For more information about these special database users, see the *Oracle7 Server Administrator's Guide*.

User Authentication

To prevent unauthorized use of a database username, Oracle provides user validation via three different methods for normal database users:

- authentication by the operating system
- authentication by a network authentication service
- authentication by the associated Oracle database

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 client/server authentication.

Because database administrators perform special database operations, Oracle requires special authentication procedures for database administrators.

Authenticating Users Using the Operating System

If your operating system permits, Oracle can use information maintained by the operating system to authenticate users. The benefits of operating system authentication are the following:

- 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, there are some special considerations with respect to distributed database environments and database links; see Chapter 21, "Distributed Databases", for information on this topic.

OSDoc

Additional Information: For more information about authenticating via your operating system, see your Oracle operating system–specific documentation.

Authenticating Users Using Network Authentication

If network authentication services, such as DCE, Kerberos, or SESAME, are available to you, Oracle can accept authentication from the network service. To use a network authentication service with Oracle, you must also have the Oracle Secure Network Services product.

If you use a network authentication service, there are some special considerations for network roles and database links. See *Oracle7 Server Distributed Systems, Volume I* for more information about network authentication.

Authenticating Users Using the Oracle Database

Oracle can authenticate users attempting to connect to a database by using information stored in that database. You must use this method when the operating system cannot be used for database user validation.

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. However, all passwords are stored in an encrypted format to maintain security for the user. A user can change his/her password at any time.

Password Encryption while Connecting

To better protect the confidentiality of your passwords, Oracle allows you to encrypt passwords during client/server and server/server connections. If you enable this functionality on the client and server machines, Oracle will encrypt passwords using a modified DES (Data Encryption Standards) algorithm before sending them across the network.

For more information about encrypting passwords in client/server systems, see *Oracle7 Server Distributed Systems, Volume I.*

Database Administrator Authentication

Database administrators must often perform special operations such as shutting down or starting up a database. Because these operations should not be performed by normal database users, the database administrator usernames need a more secure authentication scheme. Oracle provides a few methods for authenticating database administrators.

Depending on whether you wish to administer your database locally on the same machine on which the database resides or if you wish to administer many different database machines from a single remote client, you can choose between operating system authentication or password files to authenticate database administrators. Figure 17 – 1 illustrates the choices you have for database administrator authentication schemes.

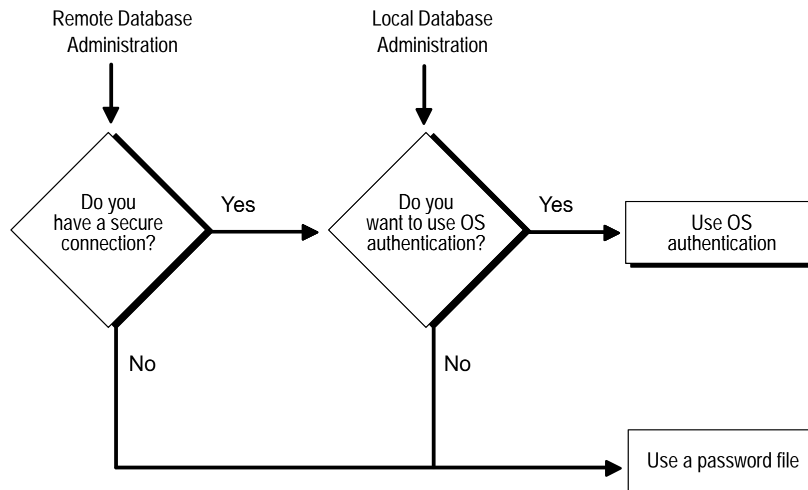


Figure 17 – 1 Database Administrator Authentication Methods

On most operating systems, OS authentication for database administrators involves placing the OS username of the database administrator in a special group (on UNIX systems, this is the dba group) or giving that OS username a special process right.

OSDoc

Additional Information: For information about OS authentication of database administrators, see your Oracle operating system–specific documentation.

Password files are files used by the database 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.

For information about password files, see the *Oracle7 Server 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 usage:

- the user's default tablespace
- the user's temporary tablespace
- space usage quotas on tablespaces of the database for the user

Default Tablespace

When a user creates a schema object and specifies no tablespace to contain the object, the object is placed in the user's default tablespace. This enables Oracle to control space usage in situations where an object's tablespace is not specified. 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

Each user can be assigned a tablespace quota for any tablespace of the database. Two things are accomplished by assigning a user a tablespace quota:

- The user can use the specified tablespace to create objects, provided that the user has the appropriate privileges.
- The amount of space that can be allocated for storage of a user's objects within the specified tablespace can be limited.

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 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, Kb, or Mb, 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 are not considered during temporary segment creation:

- Temporary segments do not consume any quota that a user might possess.
- Temporary segments can be created in a tablespace for which a user does not have a 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.

Revoking Tablespace Access

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, yet 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 (tables, views, and so on) 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. See Chapter 18, "Privileges and Roles," for more information on privileges and roles. However, to maintain tight security over access rights, grant only privileges and roles of interest to **all** users to PUBLIC.

Granting and revoking certain system and object privileges to and from PUBLIC can cause every view, procedure, function, package, and trigger in the database to be recompiled.

Restrictions for PUBLIC include the following:

- You cannot assign tablespace quotas to PUBLIC, although you can assign the UNLIMITED TABLESPACE system privilege to PUBLIC.
- You can create only database links and synonyms as PUBLIC objects, using CREATE PUBLIC DATABASE LINK/SYNONYM. No other 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 public. All rollback segments are owned by SYS. See Chapter 3, "Data Blocks, Extents, and Segments"; for more information about rollback segments.

User Resource Limits and Profiles

As part of a user's security domain, you can set limits on the amount of various system resources available to the user. By explicitly setting resource limits for each user, the security administrator can prevent the uncontrolled consumption of valuable system resources such as CPU time.

The resource limit feature of Oracle is very useful in large, multiuser systems. In such environments, system resources are very expensive; therefore, the 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 useful because users are less likely to consume system resources with detrimental impact.

You manage resource limits with user profiles. A profile is a named set of resource limits that you can assign to a user. Each Oracle database can have an unlimited number of profiles. Additionally, Oracle provides the security administrator the option to universally enable or disable the enforcement of profile resource limits.

If you use 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.

Types of System Resources and Limits

Oracle can limit the use of several types of system resources. 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. Several resource limits for Oracle can be set at the session level.

If a user exceeds a session–level resource limit, Oracle terminates the current statement (rolled back), 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 effectively 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 excessively using the system, Oracle allows several resource limits to be set 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, a certain 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 in the duration of 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 is one of the most expensive operations in a database system. I/O intensive statements can monopolize memory and disk usage and cause other database operations to compete for these resources.

To prevent single sources of excessive I/O, Oracle can 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 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, PMON cleans up after the aborted session. Until PMON completes this process, the killed session is still counted as one of the sessions for the sessions/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. The characters "K" or "M" can be used to specify Kilobytes or Megabytes.

Instructions on enabling and disabling resource limits are included in the *Oracle7 Server Administrator's Guide*.

Profiles

A profile is a named set of specified resource limits that can be assigned to valid usernames of an Oracle database. Profiles provide for easy management of resource limits.

When to Use Profiles

You only need to create and manage user profiles 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. 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 gather information about the limits CONNECT_TIME, LOGICAL_READS_PER_SESSION, and LOGICAL_READS_PER_CALL using the audit feature of Oracle. By using the AUDIT SESSION option, the audit trail gathers helpful information that you can used to determine appropriate values for the previously mentioned limits. You can gather statistics for other limits using the Monitor feature of Server Manager, specifically the Statistics monitor. The Monitor feature of Server Manager is described in the *Oracle Server Manager User's Guide*.

Licensing

Usually, Oracle is licensed for use by a maximum number of named users, or by a maximum number of concurrently connected users. The database administrator is responsible for making sure that the site complies with its license agreement. Oracle's licensing facility helps database administrators track and limit the number of sessions concurrently connected to an instance, or to limit the number of users created in a database, and thereby ensure that the site complies with the Oracle license agreement.

The database administrator controls the licensing facilities and can enable the facility and set the limits. He/she can also monitor the system's use. If the database administrator 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/she can upgrade the Oracle license to raise the appropriate limit. (To upgrade an Oracle license, you must contact your Oracle representative.)

Note: In some cases, Oracle is not licensed for either a set number of sessions or a set group of users. For example, 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 licensed differently. 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.

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 on–line users. Also, 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 database administrator to do the following:

- An administrator can 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 database administrators to kill unneeded sessions, allowing other sessions to connect.
- An administrator can 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 with RESTRICTED SESSION privilege who connects and records a warning message in the database's ALERT file.

The database administrator can set these limits in the database's parameter file so that they take effect when the instance starts. The administrator alternatively can change them while the instance is running by using the ALTER SYSTEM command. This is useful for databases that cannot be taken offline.

In addition, the session licensing mechanism allows a database administrator 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 administrator can use this information to evaluate the system's licensing needs and plan for system upgrades.

For instances running with the 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. See the *Oracle7 Server Administrator's Guide* for more information.

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 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 database administrator 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 database administrator 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 people share a user name.

The database administrator can set this limit in the database's parameter file so that it takes effect when the instance starts. The administrator can also change it while the instance is running by using the ALTER SYSTEM command. This is useful for databases that cannot be taken offline.

If multiple instances connect to the same database with the Parallel Server, all instances connected to the same database should have the same named user limit. See *Oracle7 Parallel Server Concepts & Administration* for more information.

CHAPTER

18 Privileges and Roles

My right and my privilege to stand here before you has been won —
won in my lifetime — by the blood and the sweat of the innocent.

Jesse Jackson: *Speech at the Democratic National Convention, 1988*

This chapter explains how an administrator can control users' ability to execute system operations and to access schema objects by using privileges and roles. The chapter includes:

- Privileges
- Roles

If you are configured with Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide.*

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)
- the right to create a table
- the right to select rows from another user's table
- the right to execute another user's stored procedure

Complete listings of all system and object privileges, as well as instructions for privilege management, are included in the *Oracle7 Server Administrator's Guide*.

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 lead to compromised 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
- object privileges

System Privileges

A system privilege is the right to perform a particular action, or to perform a particular action on a particular *type* of object. 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 system privileges are granted to roles, the advantages of roles can be used to manage system privileges (for example, roles permit privileges to be made selectively available).

System privileges are granted to or revoked from users and roles using either of the following:

- the Users or the Roles folders of Server Manager
- the SQL commands GRANT and REVOKE

> **Note:** Usually, you should grant system privileges only to administrative personnel and application developers because end users normally do not require the associated capabilities.

Who Can Grant or Revoke System Privileges?

Only users granted a specific system privilege with the ADMIN OPTION or users with the GRANT ANY PRIVILEGE system privilege (typically database or security administrators) can grant or revoke system privileges to other users.

Object Privileges

An object privilege is a privilege or right to perform a particular action on a *specific* table, view, sequence, procedure, function, or package. For example, the privilege to delete rows from the table DEPT is an object privilege. Depending on the type of object, there are different types of object privileges.

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.

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. 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 object remain in effect, even if the privileges were granted by specifying the dropped synonym.

Granting and Revoking Object Privileges

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 can be granted to, or revoked from, users and roles using the SQL commands GRANT and REVOKE, respectively.

Who Can Grant Object Privileges?

A user automatically has all object privileges for the objects contained in the schema that corresponds to the user's name — in other words, the schema the user owns. A user can grant any object privilege on any object he or she owns to any other user or role. If the grant includes the GRANT OPTION (of the GRANT command), the grantee can further grant the object privilege to other users; otherwise, the grantee can only use the privilege but not grant it to other users.

Table Security Topics

The object privileges for tables allow table security at two levels:

Data Manipulation Language Operations The DELETE, INSERT, SELECT, and UPDATE privileges allow the DELETE, INSERT, SELECT, and UPDATE DML operations, respectively, on a table or view (DML operations are those to view or change a table's contents). You should grant these privileges only to users and roles that need to view or manipulate a table's data. For more information on these operations, see the *Oracle7 Server SQL Reference*.

You can restrict the INSERT and UPDATE privileges for a table to specific columns of the table. With selective INSERT, a privileged user can insert a row, but only 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 could satisfy this need for additional security.

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 the privileges conservatively. In addition to these privileges, a user attempting to perform a DDL operation on a table may need other system and/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/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 Chapter 7, "Data Integrity," for more information about primary keys, unique keys, and integrity constraints.

View Security Topics

The object privileges for views allow various DML operations. Of course, a DML statement performed on a view actually affects 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 the CREATE VIEW (to create a view in your schema) or CREATE ANY VIEW (to create a view in another user's schema) system privilege, either explicitly or via a role.
- You must have been explicitly granted the SELECT, INSERT, UPDATE, and/or DELETE object privileges on all base objects underlying the view or the SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, and/or DELETE ANY TABLE system privileges. You may not have obtained these privileges through roles.

- Additionally, if you intend to grant access to your view to other users, you must have received the object privilege(s) to the base objects with the GRANT OPTION or to the system privileges with the ADMIN OPTION. If you have not, and grant access to your view, grantees cannot access your view.

Increasing Table Security Using Views To use a view, you only require the appropriate privilege for the view itself. You do not require any privileges on the base object(s) underlying the view.

Views are useful for adding two more levels of security for tables:

- A view can provide access to selected columns of the base table(s) that define the view. 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 the associated 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 base table EMP that have a salary value less than 10000. Value–based security is defined on the salary value in a row. Notice that all columns of the EMP table are accessible by the definition of the LOWSAL view.

```
CREATE VIEW own_salary AS
   SELECT ename, sal
   FROM emp
   WHERE ename = USER;
```

  The OWN_SALARY view uses the USER pseudocolumn. The values in the USER pseudocolumn are always the current user. In the OWN_SALARY view, only the rows with an ENAME that matches the user using the view are accessible. Value–based security is defined on the user accessing the view. This view combines both column–level security and value–based security.

Procedure Security Topics

The one object privilege for procedures (including standalone procedures and functions, and packages) is EXECUTE. You should grant this privilege only to users who need to execute a procedure.

You can use procedures to add a level of database security. A user requires only the privilege to execute a procedure and no privileges on the underlying objects that a procedure's code accesses. By writing a procedure and granting only the EXECUTE privilege to a user (and not the privileges on the objects referenced by the procedure), the user can be forced to access the referenced objects only through the procedure (that is, the user cannot submit ad hoc SQL statements to the database).

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.

Additionally, the user who owns the procedure must have the required privileges for the objects referenced in the body of a procedure. To create a procedure, you must have been explicitly granted the necessary privileges (system and/or object) on all objects referenced by the stored procedure; you cannot have obtained the required privileges through roles. This includes the EXECUTE privilege for any procedures that are called inside the stored 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.

Procedure Execution and Security Domains A user with the EXECUTE privilege for a specific procedure can execute the procedure. A user with the EXECUTE ANY PROCEDURE system privilege can execute any procedure in the database. A user can be granted the privileges to execute procedures via roles.

When you execute a procedure, it operates under the security domain of the user who owns the procedure, regardless of who is executing it. Therefore, a user does not need privileges on the referenced objects to execute a procedure. Because the owner of a procedure must have the necessary object privileges for referenced objects, fewer privileges have to be granted to users of the procedure and tighter control of database access can be obtained.

The **current** privileges of the owner of a stored procedure are always checked before the procedure is executed. If a necessary privilege on a referenced object is revoked from the owner of a procedure, the procedure cannot be executed by the owner or any other user.

Note: Trigger execution follows these same patterns. The user executes a SQL statement, which he/she 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.

Packages and Package Objects A user with the EXECUTE 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. Because of this, 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 encompassing package, as in the following statements:

```
GRANT EXECUTE ON hire_fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

This method of security for package objects is not discriminatory for any specific object in a package. The EXECUTE privilege granted for the package provides 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_change;

CREATE PROCEDURE hire
    BEGIN
      insert_employee(...)
    END hire;

CREATE PROCEDURE fire
    BEGIN
      delete_employee(...)
    END fire;

PACKAGE raise_bonus IS
    PROCEDURE give_raise(...) AS
      BEGIN
           change_salary(...)
      END give_raise;

    PROCEDURE give_bonus(...)
      BEGIN
           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 the top–level procedures HIRE and FIRE, and the additional package RAISE_BONUS, you can indirectly grant selective EXECUTE privileges on the procedures in the main package.

```
GRANT EXECUTE ON hire, fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

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 object privileges. However, roles are not meant to be used for application developers, because the privileges to access objects within stored programmatic constructs need to be granted directly. See the section "Data Definition Language Statements and Roles" on page 18–13 for more information about restrictions for procedures.

These properties of roles allow for easier privilege management within a database:

- *Reduced privilege administration* Rather than explicitly granting the same set of privileges 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* Because the data dictionary records which roles exist, you can design database applications to query the dictionary and automatically enable (and disable) selective roles when a user attempts to execute the application via 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. Instructions for enabling roles from an application are included in the *Oracle7 Server Application Developer's Guide.*

Common Uses for Roles

In general, you create a role to serve one of two purposes: to manage the privileges for a database application or to manage the privileges for a user group. Figure 18 – 1 and the sections that follow describe the two uses of roles.

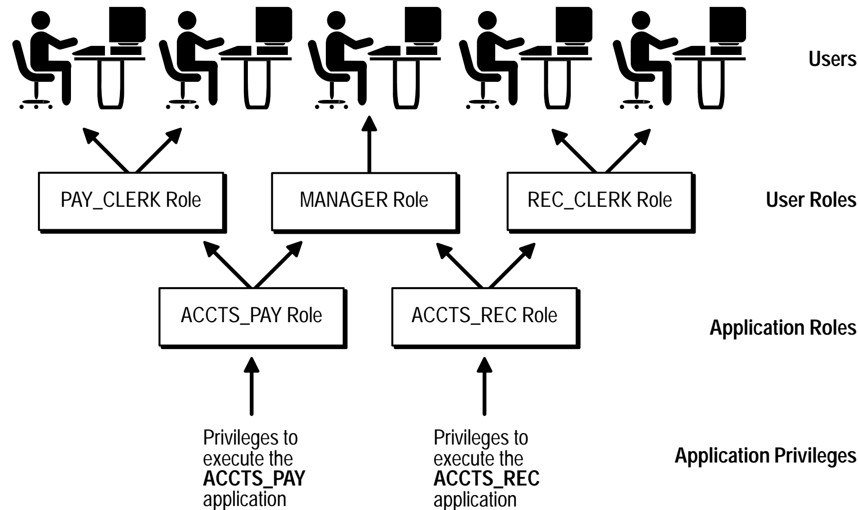


Figure 18 – 1 Common Uses for Roles

Application Roles

You grant an application role all privileges necessary to run a given database application. Then, you grant an 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

The functionality of database roles includes the following:

- A role can be granted system or 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. A user's security domain does not include 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 (a role granted to a role) 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 Add Role to User dialog box and Remove Privileges from Role dialog box of Server Manager
- the SQL commands 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.

More detailed instructions on role management are included in the *Oracle7 Server Administrator's Guide*.

Who Can Grant or Revoke Roles?

Any user with the GRANT ANY ROLE system privilege can grant or revoke *any* role to or from other users or roles of the database. You should grant this system privilege conservatively because it is very powerful. Additionally, 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.

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 a Role and a User Granted Roles

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 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 user's security domain also includes the privileges and roles granted to the user group PUBLIC. A role can be simultaneously enabled for one user and disabled for another.

Data Definition Language Statements and Roles

Depending on the statement, a user requires one or more privileges to successfully execute a DDL 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 privilege for the table or the SELECT ANY TABLE system privilege.

Oracle avoids the dependencies on privileges received via 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 object privileges that permit a user to perform a DDL operation are usable when received via a role.

  Examples System Privileges: the CREATE TABLE, CREATE VIEW and CREATE PROCEDURE privileges. Object Privileges: the ALTER and INDEX privileges for a table.

  Exception The REFERENCES object privilege for a table cannot be used for definition of a table's foreign key if the privilege is received via 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 via a role.

  Example If a user receives the SELECT ANY TABLE system privilege or the SELECT object privilege for a table via a role, he/she 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 via roles:

Example

Assume that a user

- is granted a role that has the CREATE VIEW system privilege
- is granted a role that has the SELECT object privilege for the EMP table
- is not directly granted the SELECT privilege for the EMP table
- is directly granted the SELECT object privilege for the DEPT table

Given these directly and indirectly granted privileges:

- The user can issue SELECT statements on either the EMP or DEPT tables.
- Although the user has both the CREATE VIEW and SELECT privilege for the EMP table (both via a role), the user cannot create a usable view on the EMP table, because the SELECT object privilege for the EMP table was granted via 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 (via a role) and the SELECT privilege for the DEPT table (directly).

Predefined Roles

The roles CONNECT, RESOURCE, DBA, EXP_FULL_DATABASE, and IMP_FULL_DATABASE are defined automatically for Oracle databases. 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 grants (and revokes) of database roles and/or manage their password authentication.

OSDoc

Additional Information: This capability might not be available on all operating systems. See 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 make sure that all needed roles are set as the default roles for a distributed 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 which attempts to enable a role at the remote site. To use roles in a distributed environment, you must make the required roles the default role for the remote session. For more information about distributed database environments, see *Oracle7 Server Distributed Systems, Volume I.*

CHAPTER

19

Auditing

You can observe a lot by watching.

Yogi Berra

This chapter discusses the auditing feature of Oracle. It includes:

- Introduction to Auditing
- Statement Auditing
- Privilege Auditing
- Object Auditing
- Focusing Statement, Privilege, and Object Auditing

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for additional information.

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

These sections outline 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 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, which 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, auditing only the use of the target privilege. You can set privilege auditing to audit a selected user or every user in the database.

object auditing — The selective auditing of specific statements on a particular schema object, such as AUDIT SELECT ON EMP. Object auditing is very focused, auditing only a specific statement on a specific object. Object auditing always applies to all users of the database.

You can set audit options to determine the type of audit information that is collected.

Focus of Auditing

Oracle allows audit options to be focused or broad in the following areas:

- audit successful statement executions, unsuccessful statement executions, or both
- audit statement executions once per user session or once every time the statement is executed
- audit activities of all users or of a specific user

Audit Records and the Audit Trail

Audit records include such information as the operation that was audited, the user performing the operation, and the date/time of the operation. Audit records can be stored in either a data dictionary table, called the 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 this information. Instructions for creating and using these views are included in the *Oracle7 Server Administrator's Guide.*

Depending on the events audited and the auditing options set, the audit trail records can contain different types of information. 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 object accessed
- the operation performed or attempted
- the completion code of the operation
- the date and time stamp
- the system privileges used (including MAC privileges for Trusted Oracle)
- the label of the user session (for Trusted Oracle only)
- the label of the object accessed (for Trusted Oracle only)

Audit trail records written to the OS audit trail contain some encodings that are not human readable. These can be decoded 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, while unsuccessful operations return the Oracle error code describing why the operation was unsuccessful. These codes are listed in *Oracle7 Server Messages.*

Auditing Mechanisms

These sections explain the mechanisms used by the Oracle auditing features.

When Are Audit Records Generated?

Oracle allows the recording of audit information to be enabled or disabled. This functionality allows audit options to be set by any authorized database user at any time, but reserves control of recording audit information for the security administrator. Instructions on enabling and disabling auditing are included in the *Oracle7 Server Administrator's Guide.*

Assuming auditing is enabled in the database, an audit record is generated during the execute phase of statement execution.

> **Note:** If you are not familiar with the different phases of SQL statement processing and shared SQL, see Chapter 11, "SQL and PL/SQL", for background information concerning the following discussion.

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 as INTERNAL. 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).

Events Always Audited to the Operating System Audit Trail

Regardless of whether database auditing is enabled, the Oracle Server will always audit certain database–related actions into the operating system audit trail. These events include the following:

Instance startup	An audit record is generated that details the OS user starting the instance, his terminal identifier, the date and time stamp, and whether database auditing was enabled or disabled. This is audited into the OS 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 OS user shutting down the instance, her terminal identifier, the date and time stamp.
Connections to the database as INTERNAL	An audit record is generated that details the OS user connecting to Oracle as INTERNAL. This provides accountability of users connected as INTERNAL.

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.

OSDoc

Additional Information: See 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 audit options being set or changed. A database user only adheres to modified statement or privilege audit options when the current session is ended and a new session is created. On the other hand, changes in 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 via the user account of a database link, the

remote Oracle node audits the statements issued via the database link's connection. See Chapter 21, "Distributed Databases", for more information about distributed databases and database links.

Auditing to the OS Audit Trail

Both Oracle7 and Trusted Oracle7 allow audit trail records to be directed to an operating system audit trail on platforms where the OS 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.

OSDoc

Additional Information: See your platform–specific Oracle documentation to see if this feature has been implemented on your operating system.

Trusted Oracle and Oracle allow 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 normal cause of this is that the operating system audit trail, or the file system, is full and unable to accept new records.

When configured with OS auditing, system administrators should ensure that the audit trail or the file system does not fill completely. Most operating systems provide extensive measures to provide administrators with sufficient information and warning to ensure this does not occur. Furthermore, configuring auditing to use the database audit trail removes this vulnerability, as the Oracle Server prevents audited events from occurring if the audit trail is unable to accept the 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 object, but not a specifically named structure or object (for example, AUDIT TABLE audits all CREATE and DROP TABLE statements)
- DML statements, regarding a particular *type* of database structure or object, but not a specifically named structure or 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 and audit the activities of all database users, or focused and audit only the activities of 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 object privileges are checked before the use of system privileges. If these other 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 option 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 option 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.

Privilege auditing can be broad, and audit the activities of all database users, or focused, and audit only the activities of a select list of database users.

Object Auditing

Object auditing is the selective auditing of specific DML statements (including queries), and GRANT and REVOKE statements for specific schema objects. Object auditing audits the operations permitted by 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). Notice that statements that reference clusters, database links, indexes, or synonyms are not audited directly.

You can, however, audit access to these objects indirectly by auditing the operations that affect the base table. Object audit options are always set for all users of the database; these options cannot be set for a specific list of users. Oracle provides a mechanism for setting default object audit options for all auditable schema objects.

Object Audit Options for Views and Procedures

Because views and procedures (including stored functions, packages, and triggers) reference underlying objects in their definition, auditing with respect to views and procedures has several unique characteristics. Several audit records can potentially be generated as the result of using a view or a procedure. Not only is the use of the view or procedure subject to enabled audit options, but the SQL statements issued as a result of using the view or procedure are subject to the enabled audit options of the base objects (including default audit options).

As an illustration of this situation, 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 via 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 an object invalidates that object in the cache and causes it to be reloaded. Any changes to the audit options of base 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 Object Auditing

Oracle allows statement, privilege, and object auditing to be focused in two areas:

- successful and unsuccessful executions of the audited SQL statement
- BY SESSION and BY ACCESS auditing

In addition, you can enable statement and privilege auditing for specific users or for all users in the database.

Auditing Successful and Unsuccessful Statement Executions

For statement, privilege, and 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 non–existent 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 (for example, 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 command, you can include

- the WHENEVER SUCCESSFUL option, to audit only successful executions of the audited statement
- the WHENEVER NOT SUCCESSFUL option, to audit only unsuccessful executions of the audited statement
- neither of the previous options, 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. These sections describe the distinction between the BY SESSION and BY ACCESS options of the AUDIT command.

BY SESSION

BY SESSION inserts only one audit record in the audit trail, per user and object, per session that includes an audited action. This applies regardless of whether the audit is of an object, a statement, or a privilege.

To demonstrate how the BY SESSION option allows the generation of audit records, consider the following two examples.

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 will contain 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, 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 object against which the user issued a SELECT statement in a session).

Although you can use the BY SESSION option when directing audit records to the operating system audit trail, this generates and stores an audit record each time an access is made. Therefore, in this auditing configuration, BY SESSION is equivalent to BY ACCESS.

> **Note:** A *session* is the time between when a user connects to and disconnects from an Oracle database.

BY ACCESS

Setting audit BY ACCESS inserts one audit record into the audit trail for each execution of an auditable 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 the following:

- 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 audit trail contains eight records for the eight SELECT statements.

Defaults and Excluded Operations

The AUDIT command allows you to specify either BY SESSION or BY ACCESS. However, several audit options can only be set 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 either be broad, auditing statements issued by any user, or focused, auditing statements issued by a specific list of users. By focusing on specific users, you can minimize the number of audit records generated.

PART

VIII

Distributed Processing and Distributed Databases

CHAPTER

20

Client/Server Architecture

We must try to trust one another. Stay and cooperate.

Jomo Kenyatta

This chapter defines distributed processing and 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:

- The Oracle Client/Server Architecture
- SQL*Net

The Oracle Client/Server Architecture

In the Oracle client/server architecture, the database application and the database are separated into two parts: a front–end or client portion, and a back–end or server portion. 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, it may be more efficient and effective when the client portion(s) and server portion are executed by different computers connected via a network. The following sections discuss possible variants in the Oracle client/server architecture.

> **Note:** In a distributed database, one server (Oracle) may need to access a database on another server. In this case, the server requesting the information is a client. See Chapter 21, "Distributed Databases", for more information about clients and servers in distributed databases.

Distributed Processing

Distributed processing is the use of more than one processor to divide the processing for an individual task. The following are examples of distributed processing in Oracle database systems:

- The client and server are located on different computers; these computers are connected via a network (see Figure 20 – 1, Part A).
- A single computer has more than one processor, and different processors separate the execution of the client application from Oracle (see Figure 20 – 1, Part B).

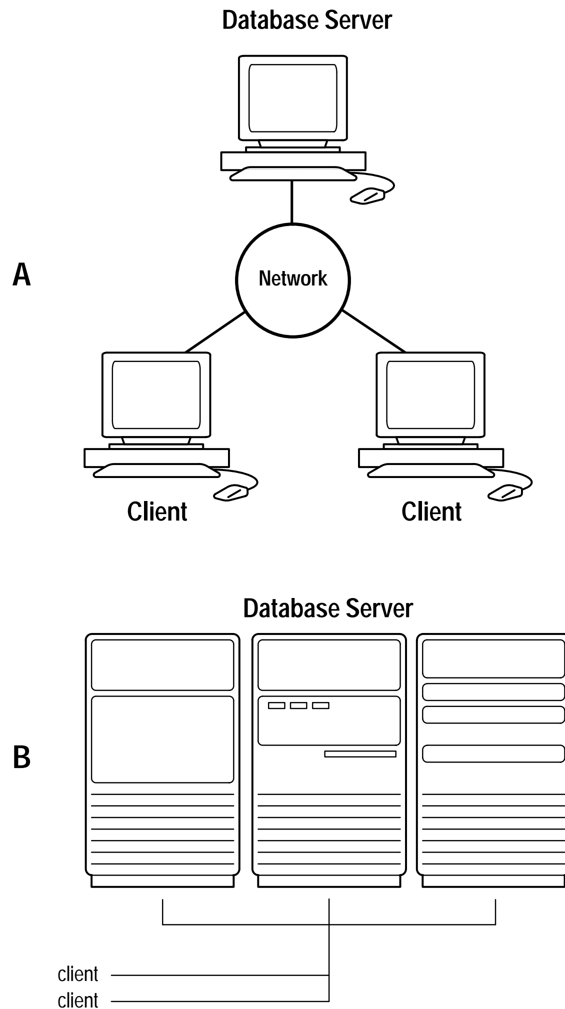


Figure 20 – 1 The Client/Server Architecture and Distributed Processing

Benefits of the Oracle client/server architecture in a distributed processing environment include the following:

- Client applications are not responsible for performing any data processing. Client applications can concentrate on requesting input from users, requesting desired data from the server, and then analyzing and presenting this data using the display capabilities of the client workstation or the terminal (for example, using graphics or spreadsheets).

- Client applications can be designed with no dependence 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).
- If necessary, Oracle can be *scaled*. As your system grows, you can add multiple servers to distribute the database processing load throughout the network (*horizontally scaled*). Alternatively, you can replace Oracle on a less powerful computer, such as a microcomputer, with Oracle running on 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, inexpensive, low–end client workstations can be used to access the remote data of the server effectively.
- 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.

SQL*Net

SQL*Net 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. SQL*Net is considered part of the program interface in network communications. See Chapter 9, "Memory Structures and Processes",.for more information about the program interface.

SQL*Net 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 communications 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 SQL*Net. SQL*Net 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 SQL*Net, 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 SQL*Net, 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.

How SQL*Net Works

SQL*Net 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 SQL*Net 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.

OSDoc

Additional Information: Depending on the operating system that executes Oracle, the SQL*Net software of the database server may include the driver software and start an additional Oracle background process; see your Oracle operating system–specific documentation for details.

For additional information on SQL*Net, refer to *Understanding SQL*Net* or the appropriate SQL*Net documentation.

CHAPTER

21 Distributed Databases

Good sense is of all things in the world the most equally distributed,
for everybody thinks he is so well supplied with it,
that even the most difficult to please in all other matters
never desire more of it than they already possess.

Rene Descartes: *Le Discours de la Methode*

This chapter describes what a distributed database is, the benefits of distributed database systems, and the Oracle distributed database architecture. The chapter includes:

- An Introduction to Distributed Databases
- Replicating Data

Note: The information in this chapter applies only for those systems using Oracle with the distributed or advanced replication options. See *Oracle7 Server Distributed Systems, Volume I* and *Oracle7 Server Distributed Systems, Volume II* for more information about distributed database systems and replicated environments.

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for information about distributed databases in that environment.

An Introduction to Distributed Databases

A *distributed database* appears to a user as a single database but is, in fact, a set of databases stored on multiple computers. The data on several computers can be simultaneously accessed and modified using a network. Each database server in the distributed database is controlled by its local DBMS, and each cooperates to maintain the consistency of the global database. Figure 21 – 1 illustrates a representative distributed database system.

The following sections outline some of the general terminology and concepts used to discuss distributed database systems.

Clients, Servers, and Nodes

A database *server* is the software managing a database, and a *client* is an application that requests information from a server. Each computer in a system is a *node*. A node in a distributed database system can be a client, a server, or both. For example, in Figure 21 – 1, the computer that manages the HQ database is acting as a database server when a statement is issued against its own data (for example, the second statement in each transaction issues a query against the local DEPT table), and 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).

Oracle supports heterogeneous client/server environments where clients and servers use different character sets. The character set used by a client is defined by the value of the NLS_LANG parameter for the client session. The character set used by a server is its database character set. Data conversion is done automatically between these character sets if they are different. For more information about National Language Support features, refer to *Oracle7 Server Reference.*

Direct and Indirect Connections

A client can connect directly or indirectly to a database server. In Figure 21 – 1, when the client application issues the first and third statements for each transaction, the client is connected directly to the intermediate HQ database and indirectly to the SALES database that contains the remote data.

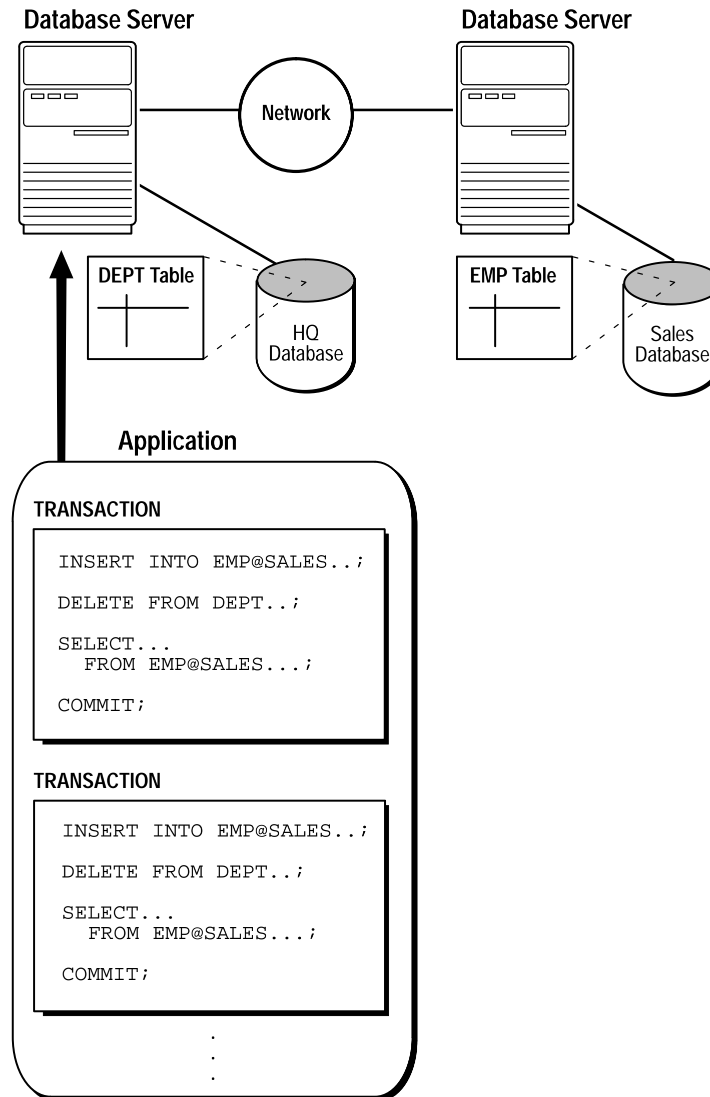


Figure 21 – 1 An Example of a Distributed DBMS Architecture

Site Autonomy

Site autonomy means that each server participating in a distributed database is administered independently (for security and backup operations) from the other databases, as though each database was a non–distributed database. Although all the databases can work together, they are distinct, separate repositories of data and are administered individually. Some of the benefits of site autonomy are as follows:

- Nodes of the system can mirror the logical organization of companies or cooperating organizations that need to maintain an "arms length" relationship.
- Local data is controlled by the local database administrator. 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. The global database is partially available as long as one database and the network are available; no single database failure need halt all global operations or be a performance bottleneck.
- Failure recovery is usually performed on an individual node basis.
- A data dictionary exists for each local database.
- Nodes can upgrade software independently.

Schema Objects and Naming in a Distributed Database

A schema object (for example, a table) is accessible from all nodes that form a distributed database. Therefore, just as a non–distributed local DBMS architecture must provide an unambiguous naming scheme to distinctly reference objects within the local database, a distributed DBMS must use a naming scheme that ensures that objects throughout the distributed database can be uniquely identified and referenced.

To resolve references to objects (a process called *name resolution*) within a single database, the DBMS usually forms object names using a hierarchical approach. For example, within a single database, a DBMS guarantees that each schema has a unique name, and that within a schema, each object has a unique name. Because uniqueness is enforced at each level of the hierarchical structure, an object's local name is guaranteed to be unique within the database and references to the object's local name can be easily resolved.

Distributed database management systems simply extend the hierarchical naming model by enforcing unique database names within a network. As a result, an object's *global object name* is guaranteed to be unique within the distributed database, and references to the object's global object name can be resolved among the nodes of the system.

For example, Figure 21 – 2 illustrates a representative hierarchical arrangement of databases throughout a network and how a global database name is formed.

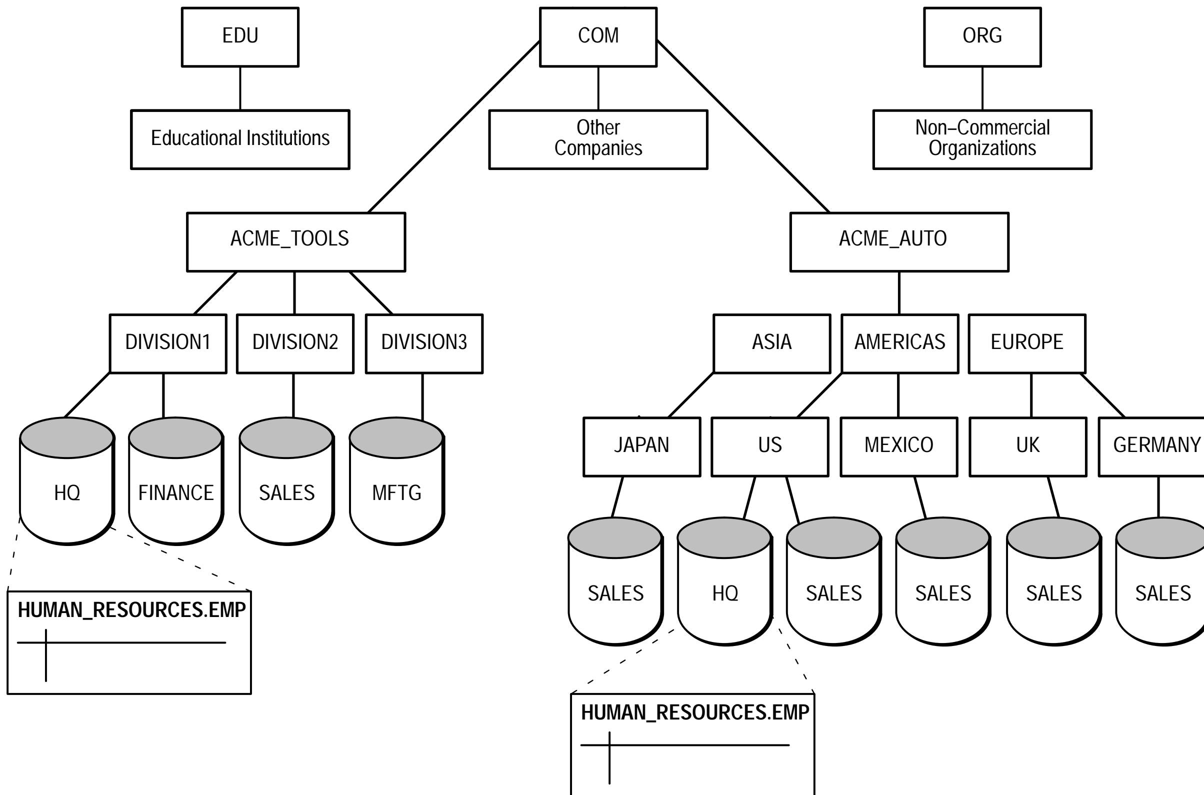


Figure 21 – 2 Network Directories and Global Database Names

Database Links

To facilitate connections between the individual databases of a distributed database, Oracle uses *database links*. A database link defines a "path" to a remote database.

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. The database link named SALES.DIVISION3.ACME.COM describes a path to a remote database of the same name.

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com ... ;
```

At this point, any application or user connected to the local database can access data in the SALES database by using global object names when referencing objects in the SALES database; the SALES database link is implicitly used to facilitate the connection to the SALES database. For example, consider the following remote query that references the remote table SCOTT.EMP in the SALES database:

```
SELECT * FROM scott.emp@sales.division3.acme.com;
```

Statements and Transactions in a Distributed Database

The following sections introduce the terminology used when discussing statements and transactions in a distributed database environment.

Remote and Distributed Statements

A *remote query* is a query that selects information from one or more remote tables, all of which reside at the same remote node.

A *remote update* is an update that modifies data in one or more tables, all of which are located at the same remote node.

> **Note:** A remote update may include a subquery that retrieves data from one or more remote nodes, but because the update is performed at only a single remote node, the statement is classified as a remote update.

A *distributed query* retrieves information from two or more nodes.

A *distributed update* modifies data on two or more nodes. A distributed update is possible using a program unit, such as a procedure or trigger, that includes two or more remote updates that access data on different nodes. Statements in the program unit are sent to the remote nodes, and the execution of the program succeeds or fails as a unit.

Remote and Distributed Transactions

A *remote transaction* is a transaction that contains one or more remote statements, all of which reference the same remote node. A *distributed transaction* is any 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. If all statements of a transaction reference only a single remote node, the transaction is remote, not distributed.

Two–Phase Commit Mechanism

A DBMS must guarantee that all statements in a transaction, distributed or non–distributed, are either committed or rolled back as a unit, so that if the transaction is designed properly, the data in the logical database can be kept consistent. The effects of a transaction should be either visible or invisible to all other transactions at all nodes; this 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 Chapter 12, "Transaction Management". 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 *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. Two–phase commit is described in Chapter 1, "Introduction to the Oracle Server".

Transparency in a Distributed Database System

The functionality of a distributed database system must be provided in such a manner that the complexities of the distributed database are transparent to both the database users and the database administrators.

For example, a distributed database system should provide methods to hide the physical location of objects throughout the system from applications and users. *Location transparency* exists if a user can refer to the same table the same way, regardless of the node to which the user connects. Location transparency is beneficial for the following reasons:

- Access to remote data is simplified, because the database users do not need to know the location of objects.
- Objects can be moved with no impact on end–users or database applications.

A distributed database system should also provide query, update, and transaction transparency. For example, standard SQL commands, such as SELECT, INSERT, UPDATE, and DELETE, should allow users to access remote data without the requirement for any programming. Transaction transparency occurs when the DBMS provides the functionality described below using standard SQL COMMIT, SAVEPOINT, and ROLLBACK commands, without requiring 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.
- The DBMS 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.

A distributed DBMS architecture should also provide facilities to transparently replicate data among the nodes of the system. Maintaining copies of a table across the databases in a distributed database is often desired so that

- Tables that have high query and low update activity can be accessed faster by local user sessions because no network communication is necessary.
- If a database that contains a critical table experiences a prolonged failure, replicates of the table in other databases can still be accessed.

A DBMS that manages a distributed database should make table replication transparent to users working with the replicated tables.

Finally, the functional transparencies explained above are not sufficient alone. The distributed database must also perform with acceptable speed.

SQL*Net and Network Independence

When data is required from remote databases, a local database server communicates with the remote database using the network, network communications software, and Oracle's SQL*Net. Just as SQL*Net connects clients and servers that operate on different computers of a network, it also connects database servers across networks to facilitate distributed transactions. For more information about SQL*Net and its features, see "SQL*Net" on page 20–5.

Heterogeneous Distributed Database Systems

The Oracle distributed database architecture allows the mix of different versions of Oracle along with database products from other companies to create a heterogeneous distributed database system.

The Mechanics of a Heterogeneous Distributed Database

In a distributed database, any application directly connected to an Oracle database can issue a SQL statement that accesses remote data in the following ways:

- Data in another Oracle database is available, no matter what version. All Oracle databases are connected by a network and use SQL*Net to maintain communication.
- Data in a non–Oracle database (such as an IBM DB2 database) is available, assuming that the non–Oracle database is supported by Oracle's gateway architecture, SQL*Connect. You can connect the Oracle and non–Oracle databases with a network and use SQL*Net to maintain communication. See the appropriate SQL*Connect documentation for more information about this product.

Figure 21 – 3 illustrates a heterogeneous distributed database system encompassing different versions of Oracle and non–Oracle databases.

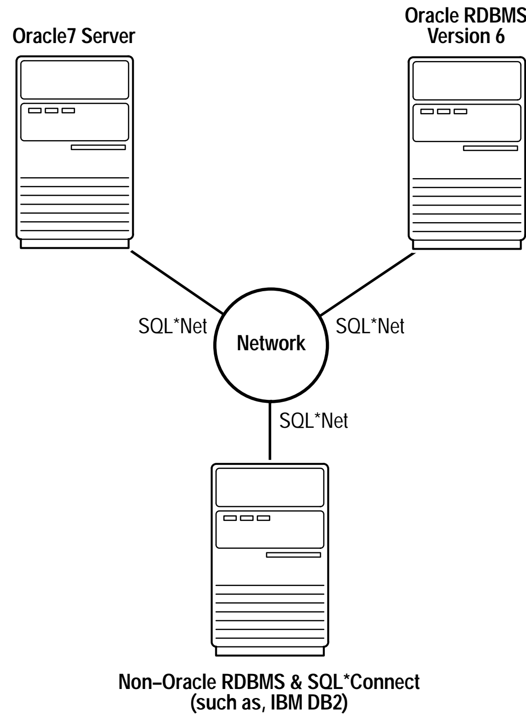


Figure 21 – 3 Heterogeneous Distributed Database Systems

When connections from an Oracle node to a remote node (Oracle or non–Oracle) initially are established, the connecting Oracle node records the capabilities of each remote system and the associated gateways. SQL statement execution proceeds, as described in the section "Statements and Transactions in a Distributed Database" on page 21–6.

However, in heterogeneous distributed systems, SQL statements issued from an Oracle database to a non–Oracle remote database server are limited by the capabilities of the remote database server and associated gateway. For example, if a remote or distributed query includes an Oracle extended SQL function (for example, an outer join), the function may have to be performed by the local Oracle database. Extended SQL functions in remote updates (for example, an outer join in a subquery) are not supported by all gateways; see your specific SQL*Connect documentation for more information on the capabilities of your system.

Replicating Data

You can create replicas of data at the various sites of a distributed database to make access to data faster for local clients. Data can be replicated using snapshots or replicated master tables. Replicated master tables require the replication option. For more information about replicating data see *Oracle7 Server Distributed Systems, Volume II.*

PART

IX

Database Backup and Recovery

CHAPTER

22 Recovery Structures

These unhappy times call for the building of plans...

Franklin Delano Roosevelt

This chapter introduces the structures of Oracle that are used during database recovery. It includes:

- An Introduction to Database Recovery and Recovery Structures
- The Online Redo Log
- The Archived Redo Log
- Control Files
- Survivability

The procedures necessary to create and maintain these structures are discussed in the *Oracle7 Server Administrator's Guide.*

An Introduction to Database Recovery and Recovery Structures

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 databases and return to normal operations as quickly as possible. Recovery should protect the databases 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, as described in Chapter 24, "Database Recovery".

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 (for example, 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 conveyed by the error message.

Process Failure

A process failure is a failure in a user, server, or background process of a database instance (for example, 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.

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 cannot continue to function correctly (usually). Therefore, you must shut down and restart the instance.

Network Failure

When your system uses networks (for example, local area networks, phone lines, and so on) 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. Once 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. Distributed database systems are discussed in Chapter 21, "Distributed Databases".

Database Instance Failure

Database instance failure occurs when a problem arises that prevents an Oracle database instance (SGA and background processes) 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.

Recovery from instance failure is relatively automatic. For example, in configurations that do not use the Oracle Parallel Server, the next instance startup automatically performs instance recovery. When using the Oracle Parallel Server, other instances perform instance recovery.

For additional information about instance recovery, see Chapter 24, “Database Recovery”.

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 physical files needed for normal database operation.

A common example of a 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, redo log files, and control files. The appropriate recovery from a media failure depends on the files affected; see Chapter 24, “Database Recovery”, for a discussion of media recovery.

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 simply 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.

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 archives filled online redo log files, Oracle returns an error in the DBWR trace file, and Oracle 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, the file is not taken offline. Instead, an error is returned and Oracle shuts

down the database. 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 filled online redo log files, DBWR fails and the current instance fails. If the problem is temporary (for example, the disk controller was powered off), 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, the entire database must be restored using the most recent backup.

Structures Used for Database Recovery

Several structures of an Oracle database safeguard data against possible failures. The following sections briefly introduce each of these structures and its role in database recovery.

Database Backups

A database backup consists of operating system backups of the physical files that constitute an Oracle database. To begin database recovery from a media failure, Oracle uses file backups to restore damaged datafiles or control files.

Oracle offers several options in performing database backups; see Chapter 23, "Database Backup", for more information.

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 be comprised of two parts: the online redo log and the archived redo log, discussed in the following sections.

The Online Redo Log Every Oracle database has an associated online redo log. The online redo log works with the Oracle background process LGWR to immediately record all changes made through the associated instance. The online redo log consists of two or more pre–allocated files that are reused in a circular fashion to record ongoing database changes; see "The Online Redo Log" on page 22–6 for more information.

The Archived (Offline) 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 and make up the archived redo log. By archiving filled online redo log files, older redo log information is preserved for more extensive database recovery operations, while the pre–allocated online redo log files continue to be reused to store the most current database changes; see "The Archived Redo Log" page 22–16 for more information.

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 (that is, 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, the data is in a consistent state after the rollback segments are used to remove all uncommitted data from the datafiles; see "Rollback Segments" on page 3–16 for more information.

Control Files

In general, the control file(s) of a database store the status of the physical structure of the database. Certain status information in the control file (for example, the current online redo log file, the names of the datafiles, and so on) guides Oracle during instance or media recovery; see "Control Files" on page 22–21 for more information.

The Online Redo Log

Every instance of an Oracle database has an associated online redo log to protect the database in case the database experiences an instance failure. An online redo log consists of two or more pre–allocated files that store all changes made to the database as they occur.

Online Redo Log File Contents

Online redo log files are filled with *redo entries.* Redo entries record data that can be used to reconstruct all changes made to the database, including the rollback segments stored in the database buffers of the SGA. Therefore, the online redo log also protects rollback data.

> **Note:** Redo entries store low–level representations of database changes that cannot be mapped to user actions. Therefore, the contents of an online redo log file should never be edited or altered, and cannot be used for any application purposes such as auditing.

Redo entries are buffered in a “circular” fashion in the redo log buffer of the SGA and are written to one of the online redo log files by the Oracle background process Log Writer (LGWR). Whenever a transaction is committed, LGWR writes the transaction’s redo entries from the redo log buffer of the SGA to an online redo log file, and a *system change number* (SCN) is assigned to identify the redo entries for each committed transaction.

However, redo entries can be written to an online redo log file before the corresponding transaction is committed. If the redo log buffer fills, or another transaction commits, LGWR flushes redo log entries in the redo log buffer to an online redo log file, of which some redo entries may not be committed. See “The Redo Log Buffer” on page 9–19 for more information.

How Online Redo Log Files Are Written

The online redo log of a database consists of two or more online redo log files. Oracle requires two files to guarantee that one is always available for writing while the other is being archived, if desired.

LGWR writes to online redo log files in a circular fashion; when the current online redo log file is filled, LGWR begins writing to the next available online redo log file. When the last available online redo log file is filled, LGWR returns to the first online redo log file and writes to it, starting the cycle again. Figure 22 – 1 illustrates the circular writing of the online redo log file. The numbers next to each line indicate the sequence in which LGWR writes to each online redo log file.

Filled online redo log files are “available” to LGWR for reuse depending on whether archiving is enabled:

- If archiving is disabled, a filled online redo log file is available once the checkpoint involving the online redo log file has completed.
- If archiving is enabled, a filled online redo log file is available to LGWR once the checkpoint involving the online redo log file has completed **and** once the file has been archived.

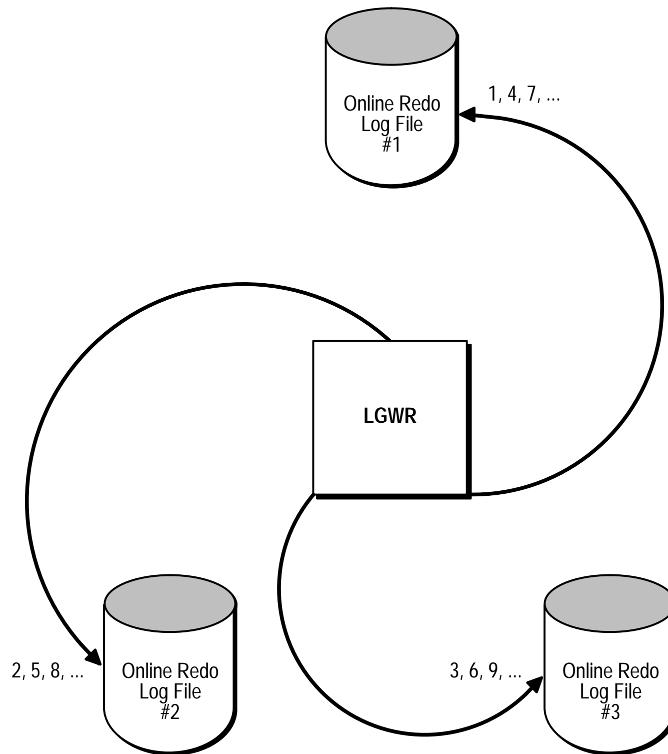


Figure 22 – 1 Circular Use of Online Redo Log Files by LGWR

Active (Current) and Inactive Online Redo Log Files

At any given time, Oracle uses only one of the online redo log files to store redo entries written from the redo log buffer. The online redo log file actively being written by LGWR is called the *current* online redo log file.

Online redo log files that are required for instance recovery are called *active* online redo log files. Online redo log files that are not required for instance recovery are called *inactive*.

If archiving is enabled, an active online log file cannot be reused or overwritten until its contents are archived. If archiving is disabled, when the last online redo log file fills, writing continues by overwriting the first available active file. For more information about archiving options for the redo log, see "Database Archiving Modes" on page 22–18.

Log Switches and Log Sequence Numbers

The point at which Oracle ends writing to one online redo log file and begins writing to another is called a *log switch.* A log switch always occurs when the current online redo log file is completely filled and writing must continue to the next online redo log file. The database administrator can also force log switches if the current redo log file is closed for some operation (for example, archiving).

Oracle assigns each online redo log file a new *log sequence number* every time that a log switch occurs and LGWR begins writing to it. If online redo log files are archived, the archived redo log file retains its log sequence number. The online redo log file that is cycled back for use is given the next available log sequence number.

Each redo log file (including online and archived) is uniquely identified by its log sequence number. During instance or media recovery, Oracle properly applies redo log files in ascending order by using the log sequence number of necessary archived and online redo log files.

Checkpoints

An event called a *checkpoint* occurs when an Oracle background process, DBWR, writes all the modified database buffers in the SGA, including committed and uncommitted data, to the data files. Checkpoints are implemented for the following reasons:

- Checkpoints ensure that data segment blocks in memory that change frequently are written to datafiles regularly. Because of the least–recently–used algorithm of DBWR, a data segment block that changes frequently might never qualify as the least recently used block and thus might never be written to disk if checkpoints did not occur.
- Because all database changes up to the checkpoint have been recorded in the datafiles, redo log entries before the checkpoint no longer need to be applied to the datafiles if instance recovery is required. Therefore, checkpoints are useful because they can expedite instance recovery.

Though some overhead is associated with a checkpoint, Oracle does not halt activity nor are current transactions affected. Because DBWR continuously writes database buffers to disk, a checkpoint does not necessarily require many data blocks to be written all at once. Rather, the completion of a checkpoint simply guarantees that all data blocks modified since the previous checkpoint are actually written to disk.

Checkpoints occur whether or not filled online redo log files are archived. If archiving is disabled, a checkpoint affecting an online redo log file must complete before the online redo log file can be reused by LGWR. If archiving is enabled, a checkpoint must complete **and** the filled online redo log file must be archived before it can be reused by LGWR.

Checkpoints can occur for all datafiles of the database (called database checkpoints) or can occur for only specific datafiles. The following list explains when checkpoints occur and what type happens in each situation:

- A database checkpoint automatically occurs at every log switch. If a previous database checkpoint is currently in progress, a checkpoint forced by a log switch overrides the current checkpoint.
- An initialization parameter, LOG_CHECKPOINT_INTERVAL, can be set to force a database checkpoint when a predetermined number of redo log blocks have been written to disk relative to the last database checkpoint. You can set another parameter, LOG_CHECKPOINT_TIMEOUT, to force a database checkpoint a specific number of seconds after the previous database checkpoint started. These options are useful when extremely large redo log files are used and additional checkpoints are desired between log switches. Database checkpoints signaled to start by these initialization parameters are not performed until the previous checkpoint has completed.
- When the beginning of an online tablespace backup is indicated, a checkpoint is forced **only** on the datafiles that constitute the tablespace being backed up. A checkpoint at this time overrides any previous checkpoint still in progress. Also, since this type of checkpoint only affects the datafiles being backed up, it does not reduce the amount of redo that would be needed for instance recovery.
- If the administrator takes a tablespace offline with normal or temporary priority, a checkpoint is forced **only** on the online datafiles of the associated tablespace.
- If the database administrator shuts down an instance (NORMAL or IMMEDIATE), Oracle forces a database checkpoint to complete before the instance is shut down. A database checkpoint forced by instance shutdown overrides any previously running checkpoint.

- The database administrator can force a database checkpoint to happen on demand. A checkpoint forced on demand overrides any previously running checkpoint.

  Note: Checkpoints also occur at other times if the Oracle Parallel Server is used; see *Oracle7 Parallel Server Concepts & Administration* for more information.

The Mechanics of a Checkpoint When a checkpoint occurs, the checkpoint background process (CKPT) remembers the location of the next entry to be written in an online redo log file and signals the database writer background process (DBWR) to write the modified database buffers in the SGA to the datafiles on disk. CKPT then updates the headers of all control files and datafiles to reflect the latest checkpoint.

When a checkpoint is not happening, DBWR only writes the least–recently–used database buffers to disk to free buffers as needed for new data. However, as a checkpoint proceeds, DBWR writes data to the data files on behalf of both the checkpoint and ongoing database operations. DBWR writes a number of modified data buffers on behalf of the checkpoint, then writes the least recently used buffers, as needed, and then writes more dirty buffers for the checkpoint, and so on, until the checkpoint completes.

Depending on what signals a checkpoint to happen, the checkpoint can be either "normal" or "fast". With a normal checkpoint, DBWR writes a small number of data buffers each time it performs a write on behalf of a checkpoint. With a fast checkpoint, DBWR writes a large number of data buffers each time it performs a write on behalf of a checkpoint.

Therefore, by comparison, a normal checkpoint requires more I/Os to complete than a fast checkpoint. Because a fast checkpoint requires fewer I/Os, the checkpoint completes very quickly. However, a fast checkpoint can also detract from overall database performance if DBWR has a lot of other database work to complete. Events that trigger normal checkpoints include log switches and checkpoint intervals set by initialization parameters; events that trigger fast checkpoints include online tablespace backups, instance shutdowns, and database administrator–forced checkpoints.

Until a checkpoint completes, all online redo log files written since the last checkpoint are needed in case a database failure interrupts the checkpoint and instance recovery is necessary. Additionally, if LGWR cannot access an online redo log file for writing because a checkpoint

has not completed, database operation suspends temporarily until the checkpoint completes and an online redo log file becomes available. In this case, the normal checkpoint becomes a fast checkpoint, so it completes as soon as possible.

For example, if only two online redo log files are used, and LGWR requires another log switch, the first online redo log file is unavailable to LGWR until the checkpoint for the previous log switch completes.

> **Note:** The information that is recorded in the datafiles and control files as part of a checkpoint varies if the Oracle Parallel Server configuration is used; see *Oracle7 Parallel Server Concepts & Administration.*

You can set the initialization parameter LOG_CHECKPOINTS_TO_ALERT to determine if checkpoints are occurring at the desired frequency. The default value of NO for this parameter does not log checkpoints. When you set the parameter to YES, information about each checkpoint is recorded in the ALERT file.

Multiplexed Online Redo Log Files

Oracle provides the capability to *multiplex* an instance's online redo log files to safeguard against damage to its online redo log files. With multiplexed online redo log files, LGWR concurrently writes the same redo log information to multiple identical online redo log files, thereby eliminating a single point of online redo log failure. Figure 22 – 2 illustrates duplexed (two sets of) online redo log files.

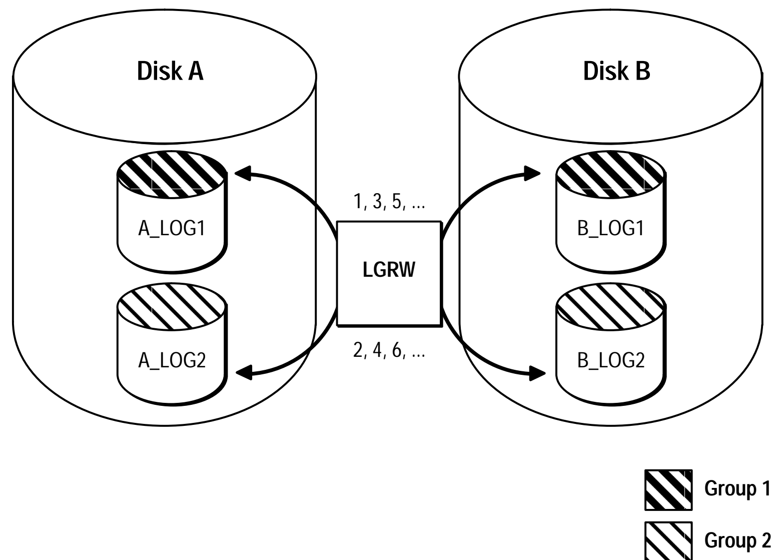


Figure 22 – 2 Multiplexed Online Redo Log Files

The corresponding online redo log files are called *groups*. Each online redo log file in a group is called a *member*. Notice that all members of a group are concurrently active (concurrently written to by LGWR), as indicated by the identical log sequence numbers assigned by LGWR. If a multiplexed online redo log is used, each member in a group must be the exact same size.

The Mechanics of a Multiplexed Online Redo Log LGWR always addresses all members of a group, whether the group contains one or many members. For example, LGWR always tries to write to all members of a given group concurrently, then to switch and concurrently write to all members of the next group, and so on. LGWR never writes concurrently to one member of a given group and one member of another group.

LGWR reacts differently when certain online redo log members are unavailable, depending on the reason for the file(s) being unavailable:

- If LGWR can successfully write to at least one member in a group (either at a log switch or as writing to the group is proceeding), writing to the accessible members of the group proceeds as normal; LGWR simply writes to the available members of a group and ignores the unavailable members.
- If LGWR cannot access the next group at a log switch because the group needs to be archived, database operation is temporarily halted until the group becomes available (in other words, until the group is archived).
- If all members of the next group are inaccessible to LGWR at a log switch because of one or more disk failures, an error is returned and the database instance is immediately shut down. In this case, the database may need media recovery from the loss of an online redo log file; see Chapter 24, "Database Recovery", for more information about such recovery. However, if the database checkpoint has moved beyond the lost log (this is not the current log in this example), media recovery is not necessary. Simply drop the inaccessible log group. If the log was not archived, archiving might need to be disabled before the log can be dropped.
- If all members of a group suddenly become inaccessible to LGWR as they are being written, an error is returned and the database instance is immediately shut down. In this case, the database might need media recovery from the loss of an online redo log file; see Chapter 24, "Database Recovery", for more information about such recovery. If the media containing the log

is not actually lost — for example, if the drive for the log was inadvertently turned off — media recovery might not be needed. In this example, all that is necessary is to turn the drive back on and do instance recovery.

Whenever LGWR cannot write to a member of a group, Oracle marks that member as stale and writes an error message to the LGWR trace file and to the database's ALERT file to indicate the problem with the inaccessible file(s).

To always safeguard against a single point of online redo log failure, a multiplexed online redo log should be completely symmetrical: all groups of the online redo log should have the same number of members. However, Oracle does not require that a multiplexed online redo log be symmetrical. For example, one group can have only one member, while other groups can have two members. Oracle allows this behavior to provide for situations that temporarily affect some online redo log members but leave others unaffected (for example, a disk failure). The only requirement for an instance's online redo log, multiplexed or non–multiplexed, is that it be comprised of at least two groups. Figure 22 – 3 shows a legal and illegal multiplexed online redo log configuration.

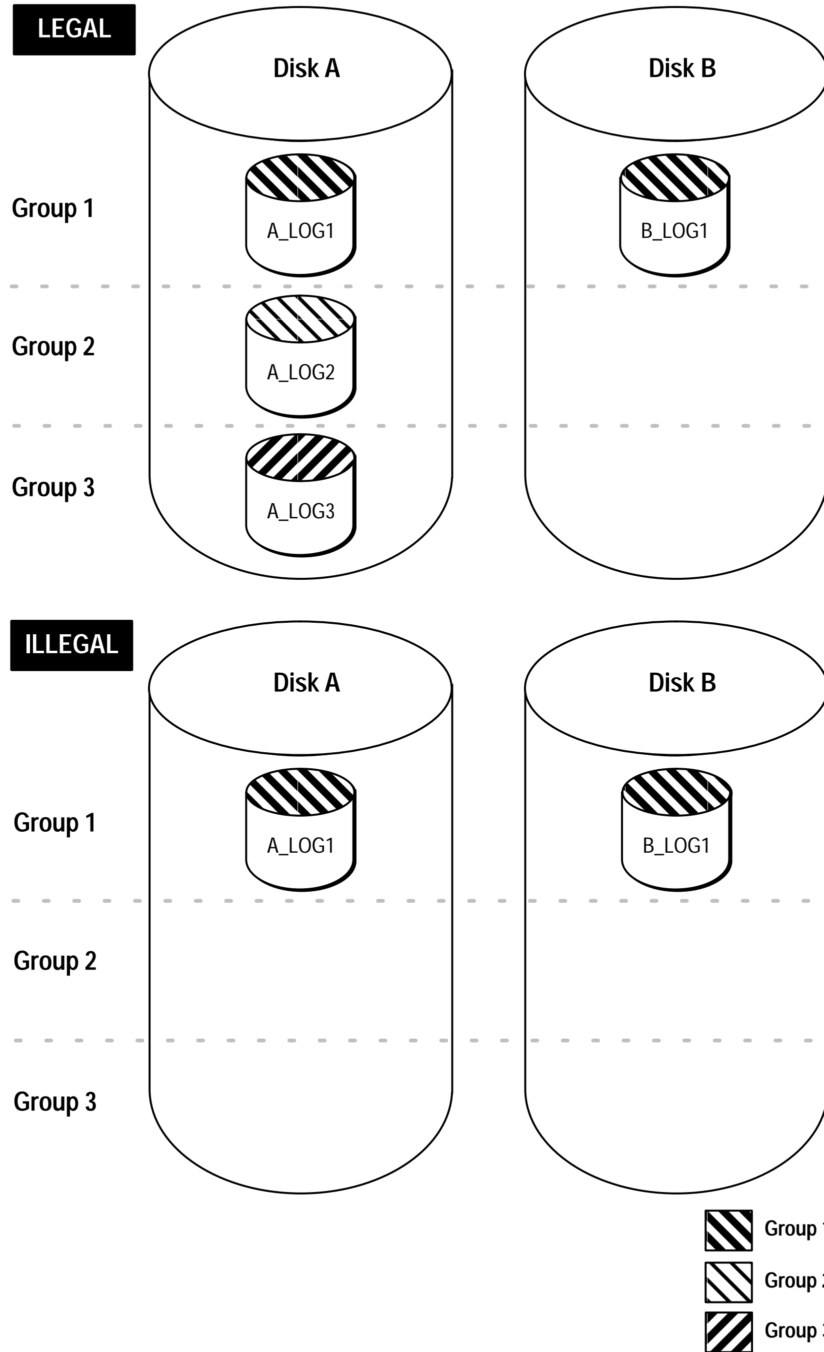


Figure 22 – 3 Legal and Illegal Multiplexed Online Redo Log Configuration

“Threads” of Online Redo Log and the Oracle Parallel Server

Each database instance has its own online redo log groups. These online redo log groups, multiplexed or not, are called an instance’s “thread” of online redo. In typical configurations, only one database instance accesses an Oracle database, thus only one thread is present. However, when running the Oracle Parallel Server, two or more instances concurrently access a single database; each instance in this type of configuration has its own thread.

This manual describes how to configure and manage the online redo log when the Oracle Parallel Server is not used. The thread number can be assumed to be 1 in all discussions and examples of commands. For complete information about configuring the online redo log with the Oracle Parallel Server, see *Oracle7 Parallel Server Concepts & Administration*.

The Archived Redo Log

Oracle allows the optional archiving of filled groups of online redo log files, which creates archived (offline) redo logs. The archiving of filled groups has two key advantages relating to database backup and recovery options:

- A database backup, together with online and archived redo log files, guarantees that all committed transactions can be recovered in the event of an operating system or disk failure.
- Online backups, taken while the database is open and in normal system use, can be used if an archived log is kept permanently.

The choice of whether or not to enable the archiving of filled groups of online redo log files depends on the backup and recovery scheme for an Oracle database. If you cannot afford to lose any data in your database in the event of a disk failure, an archived log must be present. However, the archiving of filled online redo log files can require the database administrator to perform extra administrative operations.

The Mechanics of Archiving

Depending on how you configure archiving, the mechanics of archiving redo log groups are performed by either the optional Oracle background process ARCH (when automatic archiving is used) or the user process that issues a statement to archive a group manually. For more information, see “Automatic Archiving and the ARCH (Archiver) Background Process” on page 22–19 and “Manual Archiving” on page 22–20.

> **Note:** For simplicity, the remainder of this section assumes that archiving is enabled and the ARCH background process is responsible for archiving filled online redo log groups.

ARCH can archive a group once the group becomes inactive and the log switch to the next group has completed. The ARCH process can access any members of the group, as needed, to complete the archiving of the group. For example, if ARCH attempts to open a member of a group and it is not accessible (for example, due to a disk failure), ARCH automatically tries to use another member of the group, and so on, until it finds a member of the group that is available for archiving. If ARCH is archiving a member of a group, and the information in the member is detected as invalid or a disk failure occurs as archiving proceeds, ARCH automatically switches to another member of the group to continue archiving the group where it was interrupted.

A group of online redo log files does not become available to LGWR for reuse until ARCH has archived the group. This restriction is important because it guarantees that LGWR cannot accidentally write over a group that has not been archived, which would prevent the use of all subsequent archived redo log files during a database recovery.

When archiving is used, an archiving destination is specified. This destination is usually a storage device separate from the disk drives that hold the datafiles, online redo log files, and control files of the database. Typically, the archiving destination is another disk drive of the database server This way, archiving does not contend with other files required by the instance and completes quickly so the group can become available to LGWR. Ideally, archived redo log files (and corresponding database backups) should be moved permanently to inexpensive offline storage media, such as tape, that can be stored in a safe place, separate from the database server.

At log switch time, when no more information will be written to a redo log, a record is created in the database's control file. Each record contains the thread number, log sequence number, low SCN for the group, and next SCN after the archived log file; this information is used during database recovery in Parallel Server configurations to automate the application of redo log files. See *Oracle7 Parallel Server Concepts & Administration* for additional information.

Archived Redo Log File Contents

An archived redo log file is a simple copy of the identical filled members that constitute an online redo log group. Therefore, an archived redo log file includes the redo entries present in the identical members of a group at the time the group was archived. The archived redo log file also preserves the group's log sequence number.

If archiving is enabled, LGWR is not allowed to reuse an online redo log group until it has been archived. Therefore, it is guaranteed that the archived redo log contains a copy of every group (uniquely identified by log sequence numbers) created since archiving was enabled.

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, 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, once a filled group becomes inactive and the checkpoint at the log switch completes, the group is available for reuse by LGWR.

NOARCHIVELOG mode protects a database only from instance failure, not from disk (media) failure. Only the most recent changes made to the database, stored in the groups of the online redo log, are available for instance 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 is archived. A filled group is immediately available to the process performing the archiving once a log switch occurs (when a group becomes inactive); the process performing the archiving does not have to wait for the checkpoint of a log switch to complete before it can access the inactive group for archiving. Figure 22 – 4 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, ARCH in this illustration).

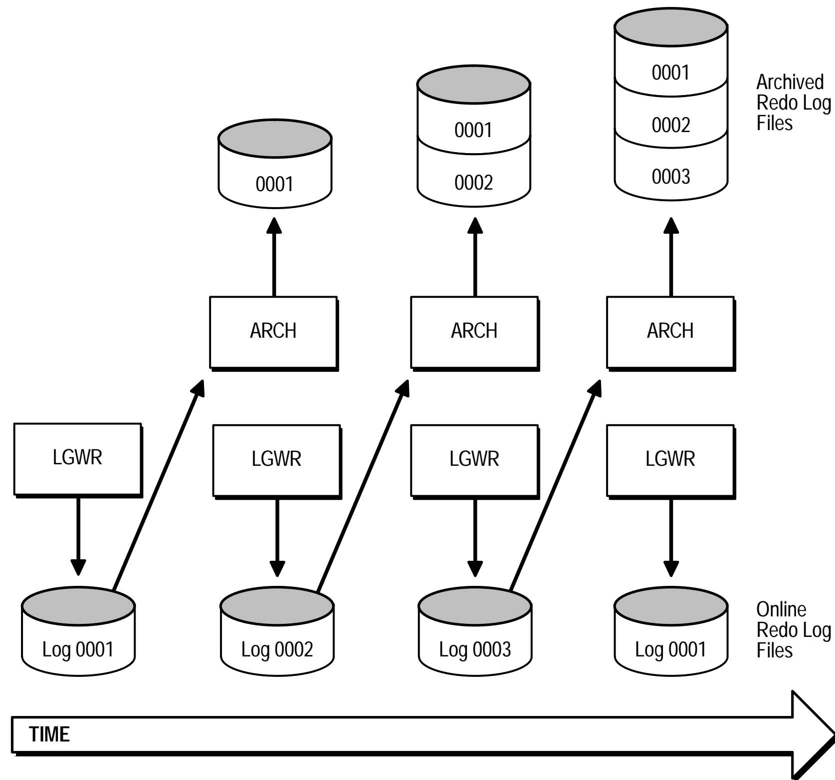


Figure 22 – 4 Online Redo Log File Use in ARCHIVELOG Mode

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.

Automatic Archiving and the ARCH (Archiver) Background Process An instance can be configured to have an additional background process, Archiver (ARCH), automatically archive groups of online redo log files once they become inactive. Therefore, 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 have an archived redo log.

If you request automatic archiving at instance startup by setting the LOG_ARCHIVE_START initialization parameter, Oracle starts ARCH during instance startup. Otherwise, ARCH is not started during instance startup.

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, the ARCH background process is created. ARCH then remains for the duration of the instance, even if automatic archiving is temporarily turned off and turned on again.

ARCH always archives groups in order, beginning with the lowest sequence number. ARCH automatically archives filled groups as they become inactive. A record of every automatic archival is written in the ARCH trace file by the ARCH process. Each entry shows the time the archive started and stopped.

If ARCH encounters an error when attempting to archive a group (for example, due to an invalid or filled destination), ARCH continues trying to archive the group. An error is also written in the ARCH 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 ARCH 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 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 archived redo log destination has experienced a failure or become full)
- archive a group in a non–standard fashion (for example, 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 ARCH 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 databases and online redo log files
- 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. See "Control File Backups" on page 23–8 for information about backing up a database's control file.

Control files also record information about checkpoints. When a checkpoint starts, the control file records information to remember the next entry that must be entered into 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 "Checkpoints" on page 22–9.

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 multiplexed control files be used with each database, with each copy stored on a different physical disk.

Survivability

In the event of a power failure, hardware failure, or any other system–interrupting disaster, Oracle7 release 7.3 offers the standby database feature. The standby database is intended for sites where survivability and disaster recovery are of paramount importance.

For information about creating and maintaining a standby database, see the *Oracle7 Server Administrator's Guide*.

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 system, you must have a plan for what to do in the event of a catastrophic failure.

See Chapter 24, "Database Recovery", for information about recovering a database.

Standby Database

Release 7.3 provides a reliable and supported mechanism for implementing a standby database system to facilitate quick disaster recovery. The scheme uses a secondary system on duplicate hardware, maintained in a constant state of media recovery through the application of log files archived at the primary site. In the event of a primary system failure, the standby can be activated with minimal recovery, providing immediate system availability. There are new commands and internal verifications for operations involved in the creation and maintenance of the standby system, improving the reliability of the disaster recovery scheme.

A standby database involves two databases: a *primary* database and a *standby* database. The primary database is the production database that is in use. The standby database is a copy of the production database, ideally located on a separate machine. The standby database runs in recovery mode until there is a failure at the primary site. At the time of a failure, the standby database performs recovery operations and comes online as the primary database.

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 standby database is therefore always behind the primary database in time and transaction history.

The physical hardware on which the standby database resides should be used only as a disaster recovery system; no other applications

should run on it. Because the standby database is designed for disaster recovery, it ideally resides in a separate physical location than the primary. The standby database exists not only to guard against power failures and hardware failures, but also to protect your data in the event of a physical disaster such as a fire or an earthquake.

CHAPTER

23 Database Backup

He listens well who takes notes.

Dante Alighieri: *The Divine Comedy*

This chapter explains the options available and the procedures necessary to backup the data in an Oracle database. It includes:

- An Introduction to Database Backups
- Read–Only Tablespaces and Backup

An Introduction to Database Backups

No matter what backup and recovery scheme you devise for an Oracle database, operating system 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. The *Oracle7 Server Administrator's Guide* provides guidelines for performing database backups.

This section includes the following topics:

- Full Backups
- Partial Backups
- The Export and Import Utilities

Full Backups

A *full backup* is an operating system backup of all datafiles and the control file that constitute an Oracle database. A full backup should also include the parameter file(s) associated with the database. You can take a full database backup when the database is shut down or while the database is open. You should not normally take a full backup after an instance failure or other unusual circumstances.

Full Online Backups vs. Full Offline Backups

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 full backup taken after a shutdown can be used to recover to the point in time of the last full backup. A full 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.

See the section "Online Datafile Backups" on page 23–4 for more information on backing up datafiles while the database is open.

Backups and Archiving Mode

The datafiles obtained from a full 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 full 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 full database 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 full backup can be used as part of database *recovery.*

  After restoring the necessary datafiles from the full 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 full backup is the only method to partially protect the database against a disk failure; if a database is operating in ARCHIVELOG mode, the files assembled by a full backup can be used to restore damaged files as part of database recovery from a disk failure.

Partial Backups

A *partial backup* is any operating system backup short of a full 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. The following sections describe each type of datafile backup.

Offline Datafile Backups Any datafile of a database can be backed up when the datafile is offline. The following situations provide examples of offline datafile backups:

- A database is shut down. As a result, all datafiles of the database are normally closed or "offline". If any datafiles of a shutdown database are backed up, these are considered offline datafile backups.
- A database is open, and a tablespace is offline. As a result, normally all datafiles of the tablespace are offline. If any datafiles of an offline tablespace are backed up, these are considered offline datafile backups.

> **Note:** In most situations, the above are examples of when offline datafile backups can be taken. However, in certain circumstances, a database may be shutdown or a tablespace can be offline, but the associated datafiles are actually online with respect to the operating system. For example, a database may be mounted and closed for database recovery, but the associated datafiles are open and undergoing changes during the recovery operation. Avoid backing up datafiles in such situations.

When a database instance is shut down in normal priority (in other words, not aborted) or when a tablespace is taken offline in normal priority (in other words, not temporary or immediate), an offline datafile backup is a copy of "consistent" data. All of the data within an offline datafile backup is consistent with respect to a single point in time — the time of the backup.

Online Datafile Backups If a database is operating in ARCHIVELOG mode, you can back up any datafile in it while the database is open, while the associated tablespace is online, and while the specific datafiles are online and currently in normal use. This type of datafile backup is considered an online datafile backup.

An online datafile backup is a copy of *fuzzy* or *inconsistent* data. A datafile that is online or being recovered is said to be "fuzzy" because the blocks are not necessarily written in the order they are changed. Therefore, all of the data within the online datafile backup is not guaranteed to be consistent with respect to a specific point in time. However, a fuzzy datafile backup is easily made consistent during database recovery procedures.

When the backup of an online tablespace (or individual datafile) starts, Oracle stops recording the occurrence of checkpoints in the headers of the online datafiles being backed up. This means that when a datafile is restored, it has "knowledge" of the most recent datafile checkpoint that

occurred *before* the online tablespace backup, not any that occurred *during* it. As a result, Oracle asks for the appropriate set of redo log files to apply should recovery be needed. Once an online backup is completed, Oracle advances the file header to the current database checkpoint.

Consistent and Fuzzy Backup Data The data in datafile backups can exist in one of two states: consistent or fuzzy.

Consistent backup data is obtained when an offline datafile is backed up. This datafile must not be offline as the result of an I/O error or have been taken offline with the immediate option. The data in a single file is said to be "consistent" with itself because all blocks of data within it correspond to a specific point in time. You still must perform recovery actions if using a consistent backup to recover a database, as the backup is consistent only with itself, not with the current point in time.

To restore datafile(s) to a particular point in time, you can use a full or partial backup taken while the database is shut down or the tablespace is offline. Because the data is already consistent, no action is required to make the data in the restored datafiles correct.

If a database is not shut down cleanly (for example, an instance failure occurred, or a SHUTDOWN ABORT statement was issued), the offline datafiles can be fuzzy.

You can also use a complete or partial database backup, taken while the database is open and the tablespace is online, to restore datafiles to a particular point in time. However, the data in the restored datafiles is fuzzy. Therefore, the appropriate redo log files (online and archived) must be reapplied to these restored datafiles to make the data consistent.

Consider the following simplified example to understand how fuzzy backup data is generated, and then used during database recovery.

Example A backup is being made of a datafile of an online tablespace. The datafile corresponds to four data blocks. For simplicity, a representative piece of data in each block is represented by a letter.

Refer to Figure 23 – 1. During this online datafile backup, the following actions take place with respect to time.

1. At the first instant in time, Block #1 of the datafile is written to the backup file.
2. At the second instant in time, Block #2 of the datafile is written to the backup file. At the same time, a modified version of data block #1 was written from the SGA to the datafile.
3. At the third instant in time, Block #3 of the datafile is written to the backup file.
4. At the fourth instant in time, a modified version of database Block #4 is written from the SGA to the datafile. This modified block is written to the backup file.

At least two redo entries were also generated because of the modifications to Blocks #1 and #4, as represented in Figure 23 – 2.

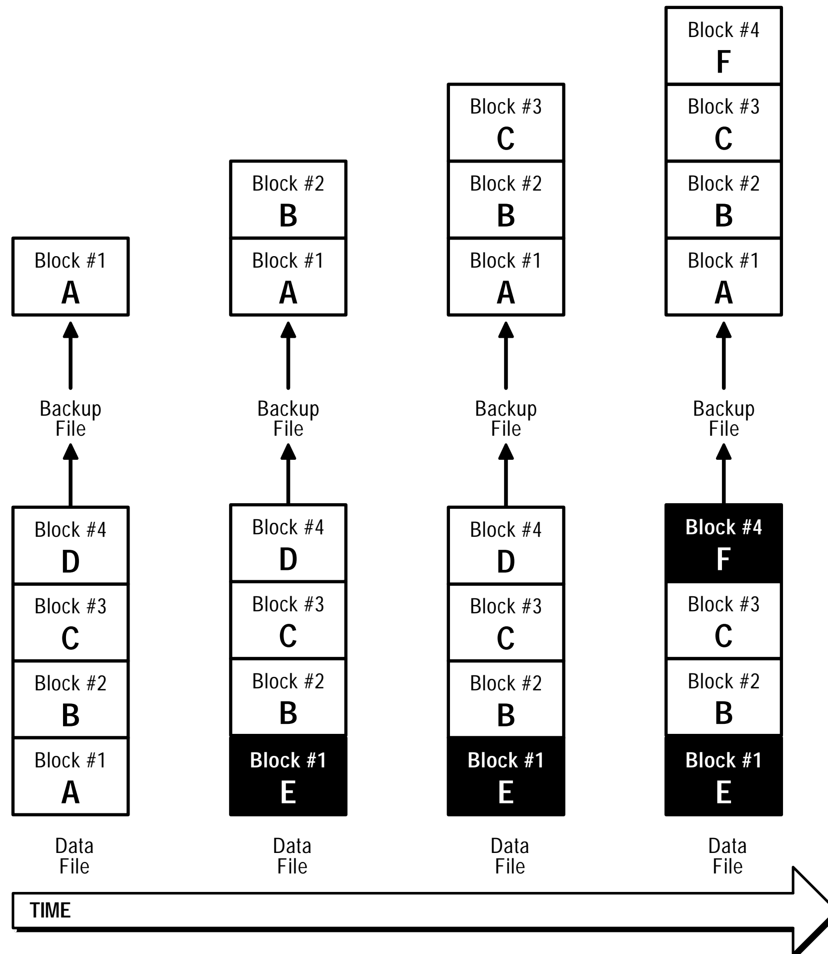


Figure 23 – 1 An Example of an Online Database File Backup

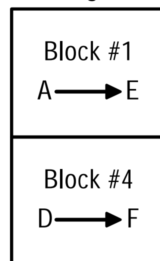


Figure 23 – 2 Redo Entires Generated During an Online Database File Backup

If the backup file restores the datafile, the restored datafile is inconsistent, as is. However, when recovery is performed the

corresponding redo log entries are checked. This causes the following actions, which make the data consistent:

1. Block #1 is updated from A to E.
2. Block #4 is verified to contain the correct information (F).

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, a backup of a database's control file should be made every time a structural change is made to the database.

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.

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. Use of these utilities is described in *Oracle7 Server 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.

Unlike backups of writeable tablespaces, you do not need to mark the beginning and end of the online backup of a read–only tablespace. Using the ALTER TABLESPACE BEGIN and END BACKUP commands with a read–only tablespace causes an error.

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 writeable, 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 writeable datafile back online.

CHAPTER

24 Database Recovery

Turn back, O man,
Forswear thy foolish ways.
Old now is Earth and none may count her days.
Da da da da da.

Steven Schwartz: *Godspell*

This chapter discusses the database recovery from instance and media failures. It includes:

- Recovery Procedures
- Recovery Features
- An Introduction to Database Recovery
- Performing Recovery in Parallel
- Recovery from Instance Failure
- Recovery from Media Failure

Recovery Procedures

In every database system, the possibility of a system failure is always present. Should system failure occur, you must recover the database as quickly, and with as little detrimental impact on users, as possible.

Recovering from any type of system failure requires the following:

1. Determining which data structures are intact and which ones need recovery.
2. Following the appropriate recovery steps.
3. Restarting the database so that it can resume normal operations.
4. Ensuring that no work has been lost nor incorrect data entered in the database.

The goal is to return to normal as quickly as possible while insulating database users from any problems and the possibility of losing or duplicating work.

The recovery process varies depending on the type of failure and the files of the database affected by the failure.

Recovery Features

Oracle offers several features to provide flexibility in recovery strategies:

- recovery from system, software, or hardware failure
- automatic database instance recovery at database start up
- recovery of individual offline tablespaces or files while the rest of a database is operational
- time–based and change–based recovery operations to recover to a transaction–consistent state specified by the database administrator
- increased control over recovery time in the event of system failure
- the ability to apply redo log entries in parallel to reduce the amount of time for recovery
- Export and Import utilities for archiving and restoring data in a logical data format, rather than a physical file backup

An Introduction to Database Recovery

The following sections provide a brief summary of how Oracle writes information to the datafiles. This discussion introduces the recovery structures and processes necessary to recover a database from any type of failure.

For instructions on performing database recovery, see the *Oracle7 Server Administrator's Guide.*

Database Buffers and DBWR

Database buffers in the SGA are written to disk only when necessary, using the least–recently–used algorithm. Because of the way that DBWR 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.
- Since the redo log might have also contained data that was not committed, the uncommitted transaction changes applied by the redo log (as well as any uncommitted changes applied by earlier redo logs) must be erased from the database.

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 and rolling back with the rollback segments.

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.* 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.

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 as well as any uncommitted changes that were recorded in the redo log.

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 recorded in the redo log and applied to the database during roll forward. This process is called *rolling back.*

Figure 24 – 1 illustrates rolling forward and rolling back, the two steps necessary to recover from any type of system failure.

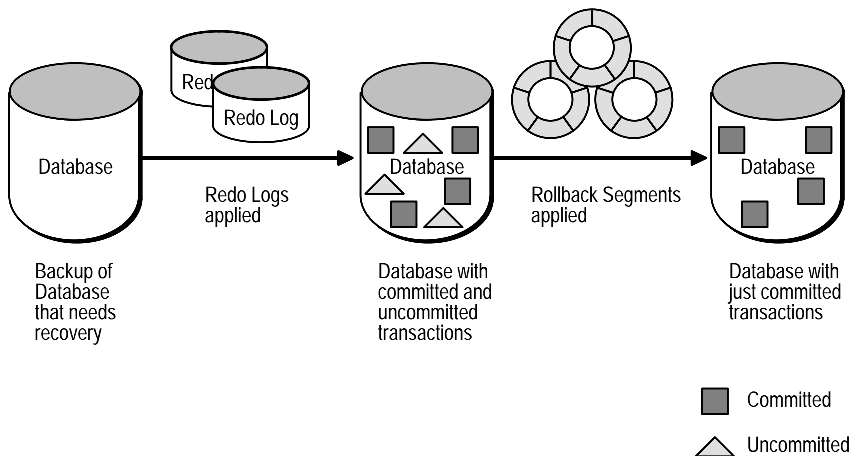


Figure 24 – 1 Basic Recovery Steps: Rolling Forward and Rolling Back

Starting with release 7.3, 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. This feature is called fast transaction rollback.

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 can simultaneously apply changes from the redo log files.

One form of parallel recovery can be performed by spawning several Server Manager sessions and issuing the RECOVER DATAFILE command on a different set of datafiles in each session. However, this method causes each Server Manager session to read the entire redo log file.

Instance and media recovery can be parallelized automatically by specifying an initialization parameter or command–line options to the RECOVER command. The Oracle Server can use one process to sequentially read the log files and dispatch redo information to several recovery processes to 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.

What Situations 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.

OSDoc

Additional Information: 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. Consult your operating system documentation to determine whether asynchronous I/O is supported on your system.

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, typically a Server Manager session or an application designed to use the ALTER DATABASE RECOVER ... command. 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 24 – 2 illustrates a typical parallel recovery session.

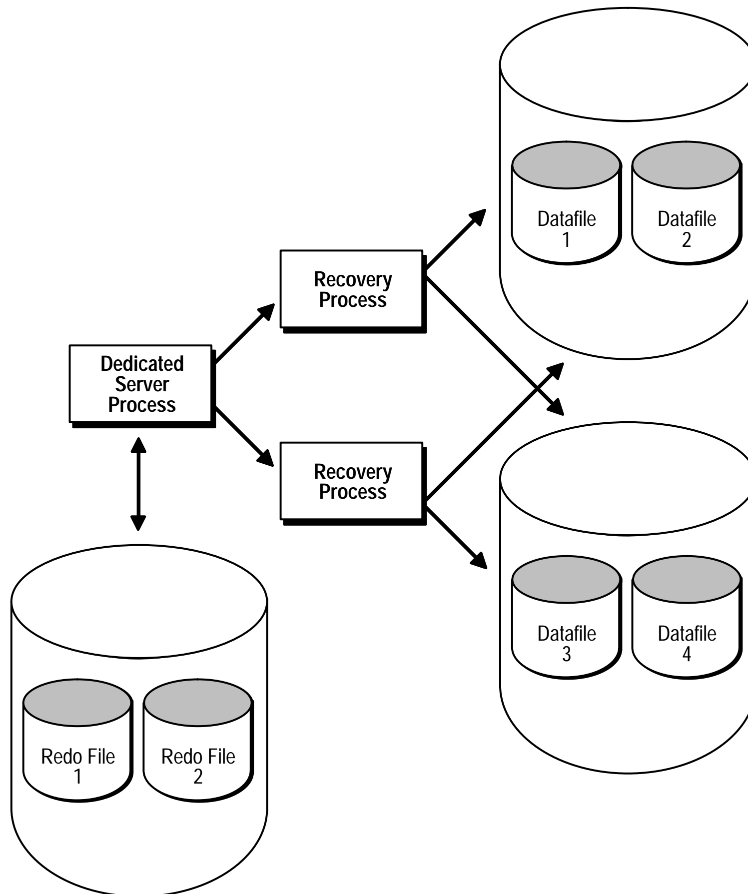


Figure 24 – 2 Typical Parallel Recovery Session

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.

Recovery from Instance Failure

When an instance is aborted, either unexpectedly (for example, an unexpected power outage or a background process failure) or expectedly (for example, when you issue a SHUTDOWN ABORT or STARTUP FORCE statement), instance failure occurs, and instance recovery is required. Instance recovery restores a database to its transaction–consistent state just before instance failure.

If you experience instance failure during online backup, media recovery might be required. In all other cases, Oracle automatically performs instance recovery for a database when the database is restarted (mounted and opened to a new instance). If necessary, the transition from a mounted state to an open state automatically triggers instance recovery, which 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.
2. Opening the database. Instead of waiting for all transactions to be rolled back before making the database available, Oracle enables the database to be opened as soon as cache recovery is complete. Any data that is not locked by unrecovered transactions is immediately available. This feature is called *fast warmstart*.
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 PARTIALLY AVAILABLE.
4. Recovering dead transactions as part of SMON recovery.
5. Resolving any pending distributed transactions undergoing a two–phase commit at the time of the instance failure.

Read–Only Tablespaces and Instance Recovery

No recovery is ever needed on read–only datafiles after instance recovery. Recovery during startup verifies that an 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, you cannot access the tablespace until you complete media recovery.

Recovery from Media Failure

Media failure is a failure that occurs when a file, portion of a file, or a disk either cannot be read from or cannot be written to because it is damaged or missing. For example, this can happen if one or more datafiles are erased accidentally or lost due to a disk head crash.

Recovery from a media failure can take two forms, depending on the archiving mode in which the database is operated:

- If a database is operated so that its online redo log is only reused and not archived, recovery from a media failure is a simple restoration of the most current full backup. All work performed after the full backup was taken must be redone manually, if desired, after the backup is used to restore the damaged database.
- If a database is operated so that its online redo log is archived, recovery from a media failure can be an actual recovery procedure, to reconstruct the damaged database to a specified transaction–consistent state before the media failure.

Recovery from a media failure, no matter what form, always recovers the entire database to a transaction–consistent state before the media failure. It is not logical or possible to recover a part of a database (such as a tablespace) to one point in time, and recover (or leave untouched) another part of a database to a different point in time; otherwise, the database would not be in a transaction–consistent state with respect to itself.

The following sections describe the different types of media recovery available if a database is operated in ARCHIVELOG mode: complete media recovery and incomplete media recovery.

Read–Only Tablespaces and Media Recovery

Normal media recovery does not check the read–only status of a datafile. When you perform media recovery of a tablespace that was once read–only, you have three possible options, depending upon when the tablespace was made read–only and when you performed the most recent backup. These scenarios are illustrated in Figure 24 – 3.

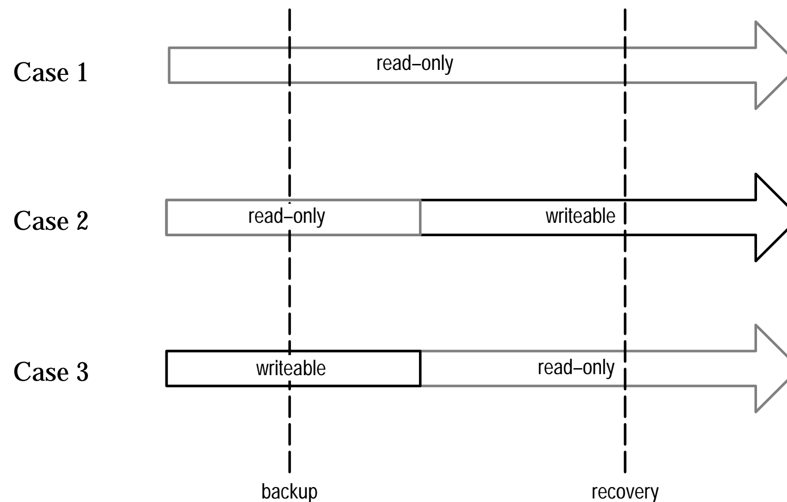


Figure 24 – 3 Type of Media Recovery

Case 1 — The tablespace being recovered is read–only, and was read–only when the last backup occurred. In this case, you can simply restore the tablespace from the backup. There is no need to apply any redo information.

Case 2 — The tablespace being recovered is writeable, but was read–only when the last backup occurred. In this case, you would need to restore the tablespace from the backup and apply the redo information from the point of time when the tablespace was made writeable.

Case 3 — The tablespace being recovered is read–only, but was writeable when the last backup occurred. Because you should always backup a tablespace after making it read–only, you should not experience this situation. However, if this does occur, you must restore the tablespace from the backup and recover up to the time that the tablespace was made read–only.

Unlike writeable datafiles, read–only datafiles are not taken offline automatically if a media failure occurs. If you experience a media failure that affects only a portion of your datafiles, you should take these datafiles offline and follow the instructions in the *Oracle7 Server Administrator's Guide* for performing recovery of offline tablespaces in an open database.

Complete Media Recovery

Complete media recovery recovers all lost changes; no work is lost. Complete media recovery is possible only if all necessary redo logs (online and archived) are available.

Different types of complete media recovery are available, depending on the files that are damaged and the availability of the database that is required during recovery operations.

Closed Database Recovery

Complete media recovery of all or individual damaged datafiles can proceed while a database is mounted but closed and completely unavailable for normal use. Closed database recovery is used in the following situations:

- The database does not have to be open (in other words, the undamaged portions of the database do not have to be available for use).
- Files damaged by the disk failure include one or more datafiles that constitute the SYSTEM tablespace or a tablespace containing active rollback segments.

Open Database–Offline Tablespace Recovery

Complete media recovery can proceed while a database is open. Undamaged tablespaces of the database are online and available for use, while a damaged tablespace is offline, and all datafiles that constitute the damaged tablespace are recovered as a unit. Offline tablespace recovery is used in the following situations:

- Undamaged tablespaces of the database must be available for normal use.
- Files damaged by the disk failure do not include any datafiles that constitute the SYSTEM tablespace or a tablespace containing active rollback segments.

Open Database–Offline Tablespace–Individual Datafile Recovery

Complete media recovery can proceed while a database is open. Undamaged tablespaces of the database are online and available for use, while a damaged tablespace is offline and specific damaged datafiles associated with the damaged tablespace are recovered. Individual datafile recovery is used in the following situations:

- Undamaged tablespaces of the database must be available for normal use.
- Files damaged by the disk failure do not include any datafiles that constitute the SYSTEM tablespace or a tablespace containing active rollback segments.

Complete Media Recovery Using a Backup of the Control File

Complete media recovery can proceed without loss of data, even if all copies of the control file are damaged by a disk failure. Media recovery of datafile backups can be done even if the control file is a backup. The control file is not recovered by media recovery; rather the RESETLOGS at database open recovers the control file.

The Mechanisms of Complete Media Recovery

The mechanisms that Oracle uses to perform any type of complete media recovery are best described using an example. The following is an example of complete media recovery of damaged datafiles while the database is open and a damaged tablespace is offline. Assume the following:

- the database has three datafiles:
  - USERS1 and USERS2 are datafiles that constitute the USERS tablespace, stored on Disk X of the database server
  - SYSTEM is the datafile that constitutes the SYSTEM tablespace, stored on Disk Y of the database server
- Disk X of the database server has crashed
- the online redo log file being written to at the time of the disk failure has a log sequence number of 31
- the database is in ARCHIVELOG mode

Recovery of the two datafiles that constitute the USERS tablespace is necessary because Disk X has been damaged, and the system has automatically taken the tablespace offline. In this case, the datafile of the SYSTEM tablespace is not damaged. Therefore, the database can be open with the SYSTEM tablespace online and available for use while recovery is completed on the offline tablespace needing recovery (USERS).

The following sections describe the phases of complete media recovery.

Phase 1: Restoration of Backup Datafiles After Disk X has been repaired, the most recent backup files are used to restore only the damaged datafiles USERS1 and USERS2. After restoration, the datafiles of the database exist as illustrated in Figure 24 – 4.

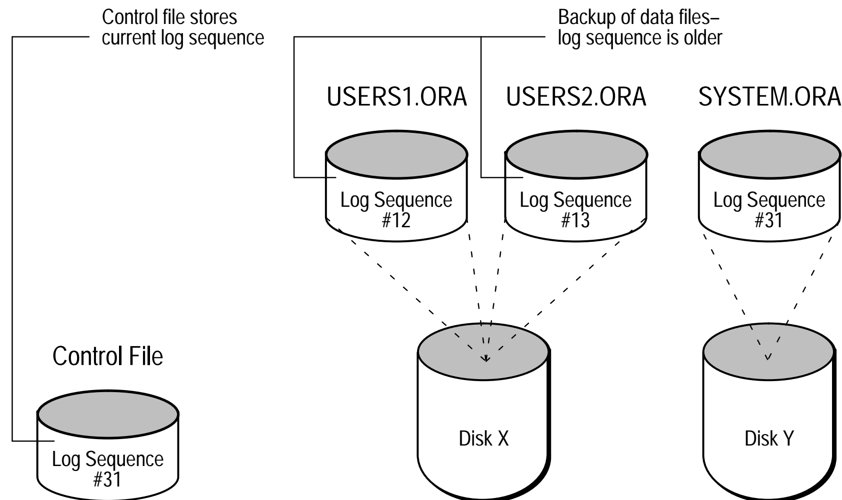


Figure 24 – 4 Phase 1 of Complete Media Recovery

Each datafile header contains the most recent log sequence number being written at the time the datafile was being written. The restored backup files will have earlier log sequence numbers than those of the datafiles that were not affected (not restored) by the disk crash. The control file contains a pointer to the last log sequence number that was written.

Phase 2: Rolling Forward with the Redo Log As complete media recovery proceeds, Oracle applies redo log files (archived and online) to datafiles, as necessary, as illustrated in Figure 24 – 5. Oracle automatically detects when a redo log file does not contain any redo information corresponding to a restored backup datafile. Therefore, Oracle optimizes the recovery process by not attempting to apply the redo log file to the restored datafile.

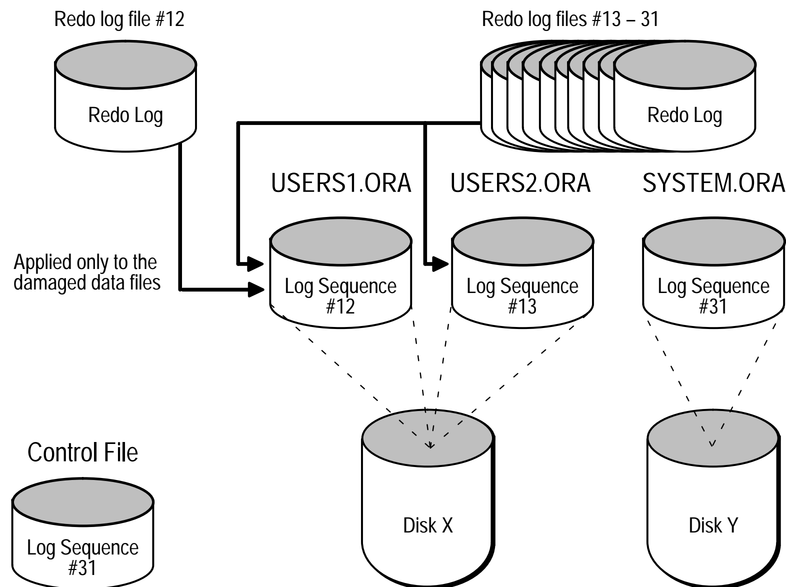


Figure 24 – 5 Phase 2 of Complete Media Recovery

In this case, the redo log file with the log sequence number of 12 is applied exclusively to USERS1, and the redo log files with log sequence numbers ranging from 13 to 31 are applied to both USERS1 and USERS2. No redo log files are applied to the datafiles that do not require recovery.

There is a flag in the header of the current redo log that indicates if it is the last available redo log file to apply to the restored datafiles.

Phase 3: Rolling Back Using Rollback Segments Once the necessary redo log files have been applied to the damaged datafiles, all uncommitted data that exists as a result of the roll forward in Phase 2 must be removed. This is completed by applying the deferred rollback segment as the tablespace is brought online, as illustrated in Figure 24 – 6.

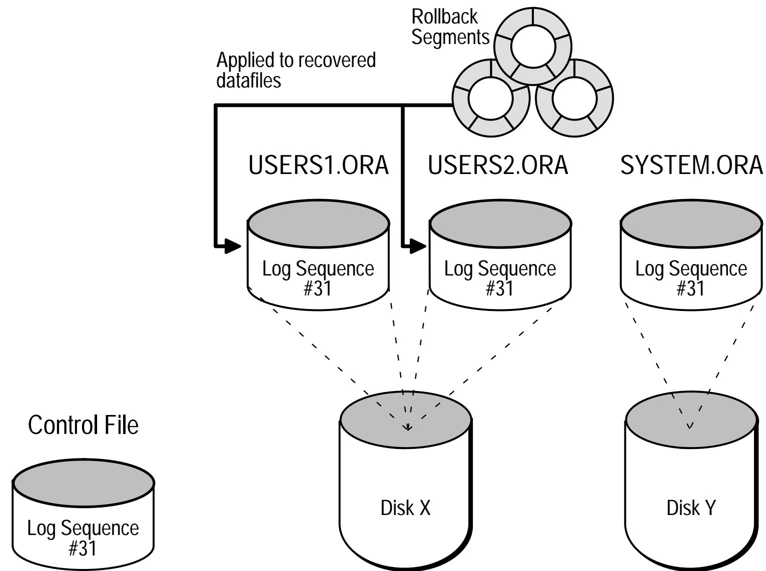


Figure 24 – 6 Phase 3 of Complete Media Recovery

After Phase 3 is complete, notice how the log sequence number contained in the datafile headers of the previously damaged and restored datafiles, USERS1 and USERS2, has been updated during Phase 2 of the recovery process. The USERS tablespace can now be brought online. Deferred rollback segments are applied to the files of the offline tablespace as it is brought back online. Once the rollback is complete, the datafiles USERS1 and USERS2 exist as they did at the instant before the disk failure. Once this is complete, all data in the tablespace is now consistent and available for use.

Incomplete Media Recovery

In specific situations (for example, the loss of all active online redo log files, or a user error, such as the accidental dropping of an important table), complete media recovery may not be possible or may not be desired. In such situations, *incomplete media recovery* is performed to reconstruct the damaged database to a transaction consistent state before the media failure or user error.

In most cases, unless desired, incomplete media recovery is not necessary if the online redo log has been mirrored to protect against having a single point of failure.

There are different types of incomplete media recovery that might be used, depending on the situation that requires incomplete media recovery: cancel–based, time–based, and change–based incomplete recovery.

Cancel–Based Recovery

In certain situations, incomplete media recovery must be controlled so that the administrator can cancel the operation at a specific point. Specifically, *cancel–based* recovery is used when one or more redo log groups (online or archived) have been damaged by a media failure and are not available for required recovery procedures (for example, the online redo log is not mirrored, and the single active online redo log file has been damaged by a disk failure). If one or more redo log groups are not available, the missing redo log groups cannot be applied during recovery procedures. Therefore, media recovery must be controlled so the recovery operation is terminated after the most recent, undamaged redo log group has been applied to the datafiles.

Time–Based and Change–Based Recovery

Incomplete media recovery is desirable if the database administrator would like to recover to a specific point in the past. This might be useful in the following situations:

- A user accidentally dropped a table and noticed the approximate time that the error was committed. The database administrator can immediately shut down the database and recover it to a point in time just before the user error.
- Part of an online redo log file (of a non–mirrored online redo log) might become corrupt due to a system failure. Therefore, the active online redo log file is suddenly unavailable, the database instance is aborted, and media recovery is now required. The redo entries in the most recently used online redo log file are valid up to the place that the corrupt data was written; later entries are invalid. Only the undamaged part of the current online redo log file can be applied. In this case, the database administrator can use time–based recovery to stop the recovery procedure once the valid portion of the most recent online redo log file has been applied to the datafiles.

In both of these cases, the endpoint of incomplete media recovery can be specified by a point in time or a specific system change number (SCN). An SCN is recorded in the redo log, along with the redo entries, each time that a transaction is committed. If a time is given, the database is recovered to the transaction consistent state just before the specified time. If an SCN is given, the database is recovered to the transaction committed just before the specified SCN.

The Mechanisms of Incomplete Database Recovery

Incomplete database recovery proceeds in the same way as complete media recovery, with a few exceptions:

- **All** datafiles must be restored using backup files completed before the intended point of recovery (the files could even come from different partial backups taken at different times). This way, the entire database is taken to a point in time before the recovery point and rolled forward to the intended point of recovery.
- For best results, the control file used during incomplete media recovery should reflect the physical structure of the database for the intended time of recovery. If you open the database using the RESETLOGS option, or if you open the database after issuing a CREATE CONTROLFILE statement, Oracle cross–checks the control file with the current data dictionary. If any datafiles have been added to, or dropped from, the data dictionary, Oracle updates the control file accordingly. Any other differences are reported with error messages.
- Recovery might terminate before all the available redo logs are applied. The recovery operation can be canceled manually, or it will be terminated automatically when the stop point is reached.
- If the database's log is reset as part of incomplete media recovery, all tablespaces containing datafiles that were offline (during the incomplete recovery) must be dropped, unless the tablespace was taken offline normally. Therefore, if you do not want to lose data corresponding to such tablespaces, restore the control file and bring the offline datafiles online before incomplete recovery.
- If an incomplete media recovery is actually a complete recovery (for example, all available redo logs were applied because a future time or SCN was specified), the database may be opened without resetting the log sequence. However, after an incomplete media recovery is finished (or complete recovery using a backup control file), the current log sequence number for the database (as noted in all datafiles and control files) must be reset to 1. This operation invalidates the redo entries present in all online redo log files and archived redo log files. After the log sequence is reset, the database's log (both online and archived) exists as if it were just created, and the online redo log files do not contain any redo entries yet. Figure 24 – 7 illustrates this concept.

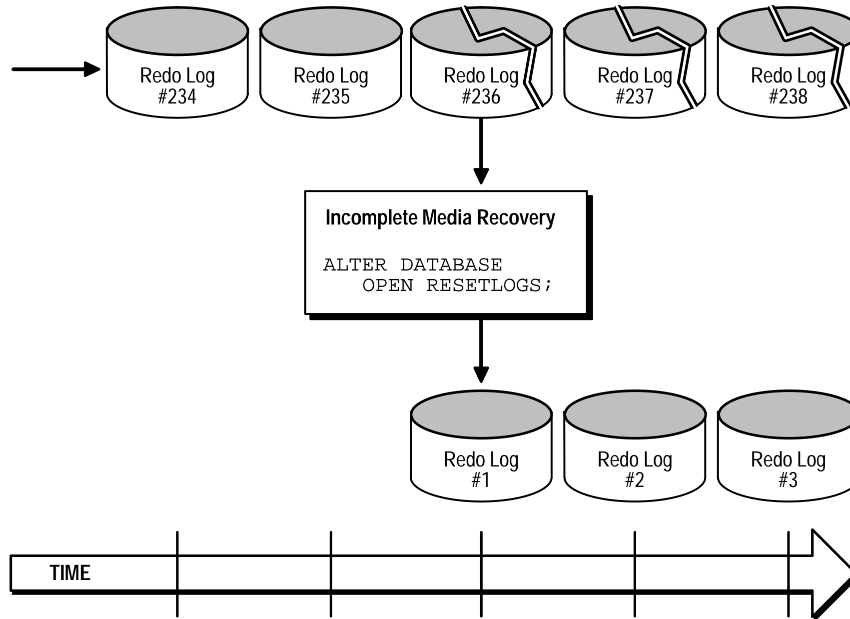


Figure 24 – 7 Effects of Resetting the Log Sequence Number after Incomplete Media Recovery

APPENDIX

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 are listed alphabetically, with page numbers of sections that discuss these topics.

- Auditing
    - operating system audit trail, 19–6
    - auditing with the OS audit trail, 19–6

- Authentication through the operating system
  - of users through operating system, 17–3
  - of database administrators, 2–2, 17–5
- Background Processes
  - creating, 9–5
  - multiple DBWR processes, 9–9
  - using ARCH, 9–12
- Client/server communication, 9–33
- Communication Software, 9–42
- Configuring Oracle, 9–30
- Datafiles
  - size of file header, 4–10
- Dedicated Server, requesting, 9–38
- Indexes, overhead of index blocks, 5–21
- INTERNAL, prerequisites for connecting as, 2–2
- Parallel recovery, 24–5
- Roles, operating system management of, 18–14
- Rollback segments, number of transactions per, 3–19
- Software code areas, shared, 9–16
- SQL*Net
  - choosing and installing drivers, 9–42
  - including drivers, 20–6

Index

A

access control
  discretionary. *See* discrete access control
  mandatory. *See* mandatory access control
  password encryption, 17–4
  privileges, 18–2
  See also privileges, roles
access paths
  defined, 13–4
  list of, 13–14
  optimization, 13–12
ADMIN OPTION, 18–3
  See also privileges; roles
AFTER triggers
  defined, 15–8
  when fired, 15–11
  See also triggers
ALERT files, 9–14
ALL_ views, 8–6
ALL_LABELS view, 6–12
ALL_UPDATABLE_COLUMNS view, 5–12
ALTER command, 10–8
ALTER TABLE ... DEALLOCATE UNUSED, 3–14
ANALYZE command
  creating histograms, 13–8
  shared pool and, 9–24
anonymous PL/SQL blocks
  about, 11–10
  vs. stored procedures, 14–8
ANSI SQL standard
  datatypes of, 6–14
  Oracle certification, 1–3
ANSI/ISO SQL standard
  data concurrency, 10–2
  isolation levels, 10–13
applications
  application triggers vs. database triggers. *See* triggers
  calling packages, 14–4
  calling procedures, 14–2
  can find constraint violations, 7–6
  data dictionary references, 8–4
  database access through, 9–29
  database links and, 21–5
  dependencies of, 16–9
    See also dependencies
  discrete transactions, 12–7
  enhancing security with, 1–37, 7–5
  network communication and, 20–5
  object dependencies and, 16–10
  procedures and, 14–8
  program interface and, 9–41
  roles and, 18–11
  sharing code, 9–16
  transaction termination and, 12–5
ARCH. *See* archiver process
architecture
  client/server, 1–45
  of Oracle. *See* Oracle, architecture
archived redo log. *See* redo log files, archived

ARCHIVELOG mode
archiver process (ARCH) and, 9–12
defined, 22–18
overview, 1–43
partial backups and, 23–3
partial database backups, 1–44
when full backups needed in, 23–3
archiver process (ARCH)
archiving online redo log files, 22–17
automatic archiving, 22–19
defined, 1–22
described, 9–12
archiving, defined, 1–43
AUDIT command, locks, 10–27
audit trail. *See* auditing, audit trail
auditing
audit options, 19–3
audit records, 19–3
audit trail, 19–3
audit trail records, 19–3
by access, 19–11
mandated for, 19–11
by session, 19–10
prohibited with, 19–11
data dictionary used for, 8–4
DDL statements, 19–6
described, 1–38 to 1–40
distributed databases and, 19–5
DML statements, 19–6
examples of, 19–8
matching database and OS usernames, 17–3
object, 19–2, 19–7
operating system audit trails, 19–5
OS audit trail, 19–6
overview of, 19–2 to 19–5
privilege use, 19–2, 19–7
range of focus, 19–3, 19–9
security and, 19–6
statement, 19–2, 19–6
successful executions, 19–9
transaction independence, 19–4
types of, 19–2
unsuccessful executions, 19–9
user, 19–11
when options take effect, 19–5
authentication
described. *See* passwords
network, 17–4
operating system, 17–3
Oracle, 17–4

B

B*–tree indexes, 5–21
back–ends, 20–2
background processes
creation of, 2–3
defined, 1–21
described, 9–5
diagrammed, 9–6
trace files for, 9–14
See also individual process names; processes
backups
consistent, 23–5
control files, 23–8
datafiles, 23–3
for read–only tablespaces, 23–8 to 23–10
full, 1–43 to 1–45, 23–2
fuzzy, 23–4
inconsistent, 23–4
online, diagrammed, 23–7
overview of, 1–40, 23–2 to 23–6
partial, 1–44, 23–3
types of, 1–43
using Export to supplement, 23–8
base tables, 1–9
BEFORE triggers
defined, 15–8
when fired, 15–11
See also triggers
blocking transactions, 10–13
blocks
anonymous, 11–7
See also data blocks; PL/SQL blocks
branch blocks. *See* indexes
buffer cache, writing of, 9–7
buffers, database. *See* database buffers

business rules
enforcing in application code, 7–5
enforcing using stored procedures, 7–5
enforcing with constraints, 1–29, 7–1
See also constraints
advantages of, 7–5 to 7–7
enforcing with triggers, 1–30
See also triggers

C

caches
buffer cache, 9–17
cache hit, 9–18
cache miss, 9–18
data dictionary, 8–4
See also data dictionary
location of, 9–20
database buffer, 1–19
See also database buffers
dictionary, 9–23
library, memory allocation for, 9–22
statistics on, 9–11
writing of buffers, 9–7

call interface. *See* Oracle call interface (OCI)

cannot serialize access, 10–13

cartesian products, 13–11

CASCADE actions, DELETE statements and, 7–15

chaining of rows, 3–10, 5–4
See also rows; data blocks

CHAR datatype
blank–padded comparison semantics, 6–3
defined, 6–2
when to use, 6–3

character sets, for various languages. *See* NLS

Check constraints
defined, 7–16
subqueries prohibited in, 7–16

check constraints, in partition views, 5–14

checkpoint process (CKPT)
defined, 9–11 to 9–13
during checkpoints, 22–11
enabling and disabling, 9–11
if not present, 9–11

checkpoints
after a time interval, 22–10
at log switches, 22–10
checkpoint process, 9–11
control files and, 22–22
database, 22–10
datafile, 22–10
DBWR process, 9–11
during online backups, 23–4
events during, 22–11
fast, 22–11
forcing, 22–11
log writer process performing, 9–11
normal, 22–11
online redo log files and, 22–10
overview of, 22–9 to 22–13
Parallel Server and, 22–11
performance effect of, 22–9
process performing (CKPT), 1–21
shutting down an instance and, 22–10
signal DBWR process, 9–8
statistics on, 9–11
taking a tablespace offline and, 22–10
types of, 22–10
when checkpoints occur, 22–10

child tables, 7–12

CKPT. *See* checkpoints, process performing

client/server architectures, 9–41
diagrammed, 20–3
direct and indirect connections, 21–2
distributed databases and, 21–2
distributed processing in, 20–3
overview of, 1–45, 20–2 to 20–5

clients
client/server architecture, 20–2
defined, 1–45

cluster keys. *See* clusters, keys

clusters
  choosing data to cluster, 5–25
  defined, 1–12
  dictionary locks and, 10–27
  hash, 5–27 to 5–34
    allocation of space for, 5–32
    collision resolution, 5–30
    contrasted with index, 5–28
    root blocks, 5–32 to 5–34
    scans of, 13–13
    storage of, 5–28
  how stored, 3–16
  index, 5–27
    contrasted with hash, 5–28
  indexes and, 5–18
  joins and, 5–25
  keys, 5–25, 5–26
    affect indexing of nulls, 5–7
  overview of, 5–23 to 5–27
  performance considerations of, 5–25
  ROWIDs and, 5–6
  scans of, 13–13
  setting parameters of, 5–26 to 5–28
  storage format of, 5–26
  storage parameters of, 5–4
columns
  correlation names, triggers use of, 15–7
  default values for, 5–7
  defined, 1–9
  described, 5–3
  integrity constraints and, 5–3, 5–7
  maximum in concatenated indexes, 5–20
  maximum in view or table, 5–9
  order of, 5–6
  prohibiting nulls in, 7–7
  USER pseudo–column, 18–6
COMMIT command, two–phase commit and, 21–6
committing transactions
  defined, 12–2
  fast commit, 9–10
  group commits, 9–10
  implementation, 9–10
  overview, 1–26
  writing the redo log buffer and, 22–7
communication protocols. *See* Network communication
compatibility, 1–4
compiled triggers, 15–14
composite indexes, 5–19
  See also indexes, composite
compression, of free space in data blocks, 3–9
concatenated indexes, 5–19
  See also indexes, composite
concurrency
  defined, 1–30
  described, 10–2
  enforced with locks, 1–32
  limits on
    per database, 17–12
    per user, 17–10
  manual locks and, 10–29
  networks and, 20–4
  restrictions on, 1–38
  transactions and, 10–17
configuration, of a database. *See* parameter files
CONNECT INTERNAL, 2–2
CONNECT role, 18–14
connectibility, 1–4, 21–8
connections
  defined, 9–30
  listener process and, 9–13
  restricting, 2–3
  sessions contrasted with, 9–30
  with administrator privileges, 2–2
consistency of data. *See* data, consistency of
consistent backups, 23–5
  See also backups
constants, in stored procedures, 11–9

constraints
  alternatives to, 7–5
  applications can find violations, 7–6
  Check, 7–16
  composite UNIQUE keys, 7–10
  default values and, 7–19
  defined, 5–3
  disabling temporarily, 7–6
  effect on performance, 7–6
  enforced with indexes, 5–20
  FOREIGN KEY, 1–30, 7–11
  mechanisms of enforcement, 7–17
  NOT NULL, 7–7
  overview, 1–29
  PRIMARY KEY, 1–29, 7–10 to 7–11
  prohibited in views, 5–9
  referential
    effect of updates, 7–15 to 7–17
    self–referencing, 7–13
  triggers cannot violate, 15–11
  triggers contrasted with, 15–5
  types listed, 1–29, 7–1
  UNIQUE key, 1–30, 7–8
  what happens when violated, 7–5
  when evaluated, 5–7, 7–17
  when triggers mandated instead, 7–4
contention
  deadlocks, 10–4
  for rollback segments. *See* rollback segments, contention for
  lock escalation does not occur, 10–5
control files
  archived redo log information in, 22–17
  backing up, 23–8
  changes recorded, 22–22
  checkpoints and, 22–22
  contents, 22–21
  defined, 1–16
  during incomplete recovery, 24–16
  how specified, 2–6
  log sequence numbers and, 22–9, 24–12
  media recovery and, 22–17
  multiplexed, 1–43, 22–22
  overview, 22–21 to 22–24
  physical database structure, 1–5
  recovery and, 1–43
  used in mounting, 2–4
  using backups of in recovery, 24–11
correlation names, triggers use of, 15–7
cost–based optimization, 13–6
CPU time limit, 17–9
CREATE PACKAGE command, locks, 10–27
CREATE PROCEDURE command, locks, 10–27
CREATE SYNONYM command, locks, 10–27
CREATE TABLE command
  locks, 10–27
  parallel query option, 3–11
CREATE TRIGGER command, locks, 10–27
CREATE VIEW command, locks, 10–27
cross joins, 13–11
cursors
  defined, 11–6
  maximum number of, 9–25
  object dependencies and, 16–7
  opening, 9–25
  overview of, 1–19
  private SQL areas and, 9–22, 11–6
  recursive, 9–25
  recursive SQL and, 9–25
  stored procedures and, 11–9

D

data
  access, procedures, 14–7
  access to, 1–24
    control of, 17–2
    security domains, 17–2
  concurrent access to, 10–2
  consistency of
    defined, 1–27
    described, 10–2
    examples of lock behavior, 10–29 to 10–36
    locks, 10–3
    manual locking, 10–29 to 10–36
    repeatable reads, 10–7
    transaction level, 10–7
    underlying principles, 10–16
  distributed manipulation of, 1–46
    See also databases, distributed; database links
  how stored in tables, 5–3

data (*continued*)
  integrity of, 1–30, 5–3, 7–2 to 7–4
    See also constraints
    Check constraints, 7–16
    comparison of enforcement methods, 7–5 to 7–7
    enforcing with triggers, 7–4
    overview, 1–28
    read consistency, 1–31
    referential, 7–3
      See also referential integrity
    two–phase commit, 1–46
    types, 7–2 to 7–4
  locks on, 10–19 to 10–25
  replicating, 1–47, 21–10
  storage of in clusters, 5–4
data access. *See* data, access to
data blocks, 1–13
  allocating for extents, 3–13
  cached in memory, 9–8
  clustered, 5–26
  coalescing free, 3–13
  controlling free space in, 3–5
  format, 3–3 to 3–8
  free lists and, 3–9
  hash keys and, 5–32 to 5–34
  header's row directory, 5–6
  how rows stored in, 5–4
  overview, 3–2 to 3–11
  read–only transactions and, 10–29
  shared in clusters, 5–23
  shown in ROWIDs, 6–9
  space available for inserted rows, 3–9
  stored in the buffer cache, 9–17
  writing to disk, 9–8
Data Definition Language (DDL)
  auditing, 19–6
  defined, 1–24
  described, 11–4
  locks, 10–26
data dictionary
  access to, 8–3
  adding objects to, 8–4
  ALL prefixed views, 8–6
  audit trail (SYS.AUD$), 8–4
  cache, 9–23
    location of, 9–20
data dictionary (*continued*)
  content of, 9–23
  DBA prefixed views, 8–6
  defined, 1–16
  dependencies tracked by, 16–3
  DUAL table, 8–6
  dynamic performance tables, 8–7
  locks, 10–26
  overview of, 8–2 to 8–4
  owner of, 8–3
  prefixes to views of, 8–5
  public synonyms for, 8–5
  row cache and, 9–23
  structure of, 8–3
  updates of, 8–4
  USER prefixed views, 8–5
  uses of, 8–3
  views used in optimization, 13–7
data locks
  conversion, 10–17
  duration of, 10–17
  escalation, 10–17
Data Manipulation Language (DML)
  auditing, 19–6
  defined, 1–25
  described, 11–3
  locks acquired by, 10–24
  privileges controlling, 18–4
  triggers' use of, 15–13
Data manipulation statements (DML), allowed in distributed transactions, 21–6
data models, 1–7
data segments, 3–16, 5–3
  See also segments
database administrators (DBA's)
  authentication, 17–4 to 17–7
  connecting with administrator privileges, 2–2
  data dictionary views of, 8–6
  password files, 17–5
  responsibilities for backup, 22–2

database buffers
  after committing transactions, 12–5
  buffer cache, 9–7, 9–17
  checkpoints and, 22–9
  clean, 9–7
  committing transactions, 9–10
  defined, 1–19
  dirty, 9–7, 9–17
  free, 9–17
  pinned, 9–17
  size of cache, 9–18
  writing of, 9–7

database links
  defined, 1–12
  overview of, 21–5 to 21–10
  See also databases, distributed

database management systems (DBMS's), principles, 1–7 to 1–9

database management systems (DBMSs)
  Oracle Server, 1–4
  See also databases

database triggers. *See* triggers

database writer process (DBWR)
  checkpoints and, 22–9
  checkpoints signal, 9–8
  defined, 9–7
  least recently used algorithm (LRU), 9–8
  multiple, 9–9
  overview of, 1–21
  time–outs, 9–9
  when active, 9–8
  writing to disk at checkpoints, 9–11

databases
  access control, password encryption, 17–4
  accessing non–Oracle, 21–8
  backing up, 1–43 to 1–45, 23–2 to 23–6
  closing, 2–5
    aborting the instance, 2–5
  configuring, 2–6 to 2–8
  contain schemas, 17–2
  defined, 1–7
  dismounting, 2–6
  distributed
    See also distributed databases
    local, 1–46
    nodes of, 1–46
    overview of, 1–45 to 1–49
    remote, 1–46
    site autonomy of, 21–3
    statement optimization on, 13–18
    two–phase commit, 1–46
  global database names, changing, 9–25
  implementation of, 4–2 to 4–4
  logical structure of, 1–5
  logical structures (objects) in, 1–8
  modes of
    ARCHIVELOG, 22–18
    NOARCHIVELOG, 22–18
  mounting, 2–3 to 2–5
  open and closed, 2–2
  opening, 2–4 to 2–6
    acquiring rollback segments and, 3–23
  physical structure, 1–5, 1–15 to 1–18, 3–2 to 3–29
  physical structure of, revealing with ROWIDs, 6–10 to 6–12
  recovery of, 1–40
    See also recovery
  scaled, 20–4
  shutting down, 2–5
  size of
    enlarging, 4–4
    how determined, 4–6
  standby, 22–23
  starting up, 2–2
    forced, 2–6
  usage of, limitations on, 17–8

datafiles
  adding to tablespaces, 4–4 to 4–6
    cannot if read–only, 4–9
  backing up, 23–3
  backups, media recovery and, 24–11
  contents of, 4–10
  in online or offline tablespaces, 4–11
  log sequence numbers and, 24–12
  named in control files, 22–21
  offline during incomplete recovery, 24–16
  overview, 1–15 to 1–17
  overview of, 1–8 to 1–10, 4–10 to 4–12
  physical database structure, 1–5
  read–only, 4–8
    recovery, 24–7
  recovery of, 24–10
  relationship to tablespaces, 4–2

datafiles (*continued*)
  shown in ROWIDs, 6–9
  taking offline, 4–11
datatypes
  ANSI, 6–14
  CHAR, comparisons of values, 6–3
  character, 6–2
  choosing a character datatype, 6–3
  conversions of
    Oracle to another Oracle type, 6–15 to 6–19
    Oracle to non–Oracle types, 6–14, 9–41
  DB2, 6–14
  how they relate to tables, 5–3
  list of available, 6–2
  LONG, storage of, 5–6
  numeric, 6–4
  of columns, 1–9
  SQL/DS, 6–14
  VARCHAR2, comparison of values, 6–3
  See also individual datatype names
DATE datatype
  arithmetic with, 6–7
  changing default format of, 6–6
  described, 6–6
  Julian dates, 6–6
DB_BLOCK_BUFFERS parameter
  buffer cache and, 9–18
  system global area size and, 9–26
DB_BLOCK_LRU_LATCHES parameter, 9–8
DB_BLOCK_SIZE parameter, System Global Area size and, 9–26
DB_FILE_MULTIBLOCK_READ_COUNT parameter, 13–17
DB_FILES parameter, 9–28
DB_WRITERS, 9–9
DBA role, 18–14
DBA_ views, 8–6
DBA_SYNONYMS.SQL script, using, 8–6
DBA_UPDATABLE_COLUMNS view, 5–12
DBMS. *See* databases
DBMS_SQL package, 11–10 to 11–12
DBWR. *See* database writer process
DDL. *See* Data Definition Language
deadlocks
  artificial, 9–37
  avoiding, 10–18
  defined, 10–4
  detection of, 10–17
  distributed transactions and, 10–18
deallocating extents, 3–14
dedicated servers, 9–32
  defined, 1–20
  examples of use, 9–39
  multi–threaded servers contrasted with, 9–34
default values, 5–7
  See also columns; datatypes
  constraints effect on, 5–7, 7–19
delete cascade, 7–15
DELETE command
  foreign key references and, 7–15
  freeing space in data blocks, 3–9
delete restrict, 7–15
dependencies
  between schema objects, 16–2
    See also schema objects, dependencies
  local, 16–8
  non–existent referenced objects and, 16–6
  on non–existence of other objects, 16–6
  Oracle Forms triggers and, 16–10
  privileges and, 16–6
  remote objects and, 16–8
  shared pool and, 16–7
dependent tables, 7–12
dictionary cache locks, 10–28
different row–writers block writers, 10–13
dirty buffer, 9–17
dirty read, 10–3, 10–13
dirty write, 10–13
disaster recovery, 22–23
discrete transaction management, summary, 12–7
discretionary access control, 1–34, 17–1
disk failures. *See* failures, media

disk space
controlling allocation for tables, 5–3
datafiles used to allocate, 4–10
See also datafiles
dispatcher processes (Dnnn)
defined, 1–22
described, 9–13
limiting SGA space per session, 17–11
listener process and, 9–13
network protocols and, 9–13
prevent startup and shutdown, 9–38
response queue and, 9–37
user processes connect through SQL*Net, 9–13
distributed databases
auditing and, 19–5
client/server architectures and, 20–2
database links, 21–5 to 21–10
deadlocks and, 10–18
dependent objects and, 16–8
diagrammed, 21–2
distributed queries, 21–6
distributed updates, 21–6
global object names, 21–4
heterogeneous, 21–8 to 21–11
nodes of, 21–2
overview of, 1–45 to 1–49, 21–2 to 21–9
performance and, 21–8
procedure dependencies in, 16–9
recoverer process (RECO) and, 9–12 to 9–14
referential integrity and, 7–4
remote queries and updates, 21–6
server can also be client in, 20–2
site autonomy of, 21–3
snapshot refresh process (SNPn), 9–13
SQL*Net, 21–8
statement optimization on, 13–18 to 13–20
table replication, 1–47
transparency of, 21–7
two–phase commit, 1–46
See also databases, distributed; dependencies
distributed processing environment
client/server architecture in, 20–3
described, 20–2
distributed transactions
defined. *See* transactions, distributed
in a heterogeneous environment, 21–9
involving non–Oracle databases, 21–8
DISTRIBUTED_TRANSACTIONS parameter, 9–12
DML. *See* Data Manipulation Language
Dnnn. *See* dispatcher processes
drivers, 9–42
DUAL table, 8–6
dynamic performance tables (V$ tables), 8–7
dynamic SQL, DBMS_SQL package, 11–10 to 11–11

E

embedded SQL statements, 1–25, 11–5
equijoins, defined, 13–11
errors
during PL/SQL compilation, 14–15
internal, tracked in trace files, 9–14
exceptions
during trigger execution, 15–12
raising, 11–10
stored procedures and, 11–9
exclusive locks
DDL locks, 10–26
defined, 10–4
row locks (TX), 10–20
RX locks, 10–22
table locks (TM), 10–20
exclusive mode, 2–4
See also Parallel Server, exclusive mode
execution plans
execution sequence of, 13–4
location of, 9–21
overview of, 13–2 to 13–5
viewing, 13–5
EXP_FULL_DATABASE role, 18–14
explicit locking, 10–29 to 10–36
See also locks
Export utility, use in backups, 23–8

extents
  allocating data blocks for, 3–13
  allocation to rollback segments
    after segment creation, 3–21
    at segment creation, 3–19
  allocation, how performed, 3–13
  as collections of data blocks, 3–10
  deallocation, when performed, 3–14
  deallocation from rollback segments, 3–22
  defined, 3–3
  dropping rollback segments and, 3–23
  in rollback segments, changing current, 3–19
  incremental, 3–11
  initial, 3–10
  overview of, 3–10 to 3–14
  parallel query option, 3–11

F

failures
  archiving redo log files, 22–20
  database buffers and, 24–3
  database instance, 1–41
  described, 22–2
  during checkpoints, 22–11
  instance, 22–3
    recovery from, 24–7
  internal errors, tracked in trace files, 9–14
  media
    described, 22–4
    multiplexed online redo logs and, 22–12
    recovery from, 24–8 to 24–17
  network, 22–3
  safeguards provided, 22–5
  statement and process, 1–40
  types of, 1–40
  user error, 1–40, 22–2
fast commit, 9–10
fast transaction rollback, 24–4
fast warmstart, 24–7
file management locks, 10–28
files
  operating system, 1–5
  X, 7–2
FIPS standard, 1–3
flagging of nonstandard features, 1–3
FOREIGN KEY constraints
  changes in parent key values, 7–15
  deleting parent table rows and, 7–15
  maximum number of columns in, 7–12
  nulls and, 7–14
  updating parent key tables, 7–15
foreign keys
  See also data, integrity of
  defined, 1–30
  partially null, 7–14
  privilege to use parent key, 18–5
free lists, 3–9 to 3–11
free space (section of data blocks), 3–5
front–ends, 20–2
full backups, 23–2
full table scans, 13–13
functions
  contrasted with procedures, 1–28
  described, 14–2
  hash, 5–31 to 5–32
  in views, 5–11
  packages can contain, 14–3
  privileges for, 18–7
  procedures contrasted with, 14–6
fuzzy backups, 23–4
  See also backups
fuzzy reads, 10–3

G

gateways, 21–9
global object names, 1–12
GRANT ANY PRIVILEGE system privilege, 18–3
GRANT command, locks, 10–27
granting, privileges and roles. *See* privileges, granting; roles, granting
group commits, 9–10

H

handles, for SQL statements. *See* cursors
hash clusters
  overview of, 1–12
  See also clusters; clusters, hash
HASHKEYS parameter, 5–30
headers
  of data blocks, 3–4
  of row pieces, 5–4
    See also data blocks; rows
HIGH_VALUE column, of USER_TAB_COL-UMNS view, 13–18
histograms, 13–8

I

IMP_FULL_DATABASE role, 18–14
Import utility, use in recovery, 23–8
in–doubt transactions. *See* transactions, in–doubt
inconsistent, backups, 23–4
index segments, 1–14, 3–16
indexes
  B*–tree structure of, 5–21 to 5–23
  branch blocks, 5–22
  cluster, 5–27
    contrasted with table, 5–27 to 5–29
    dropping, 5–27
  composite, 5–19
  concatenated, 5–19
  described, 1–11
  internal structure of, 5–21 to 5–23
  keys and, 5–20
  leaf blocks, 5–22
  location of, 5–20
  LONG RAW datatypes prohibit, 6–8
  non–unique, 5–19
  nulls and, 5–7
  overview of, 5–18 to 5–23
  parallel query option, 3–11
  performance and, 5–18
  PRIMARY KEY constraints enforced with, 7–11
indexes (*continued*)
  ROWIDs and, 5–22
  scans of, 13–13
  storage format of, 5–21
  unique, 5–19
  UNIQUE key constraints enforced with, 7–9
  when used with views, 5–11
INIT.ORA files. *See* parameter files
initialization parameters, DB_BLOCK_LRU_LATCHES, 9–8
INSERT command
  foreign key references and, 7–15
  free lists and, 3–9
instance recovery. *See* recovery, instance
instances
  acquire rollback segments, 3–23
  associating with databases, 2–3, 2–6
  configuration of, 9–29 to 9–38
  defined, 1–19
  described, 9–2
  diagrammed, 9–7
  failure in, 1–41, 22–3
  memory structures of, 9–15
  multi–process, configurations of, 9–29
  multiple–process, 9–4
  overview of, 1–5
  recovery of
    automatic, 9–11
    procedure, 24–7 to 24–9
  restricted mode, 2–3
  sharing databases, 1–6
    See also Parallel Server
  shutting down, 2–5, 2–6
  single–process, 9–3
  starting, 2–3
  virtual memory, 9–15
integrity constraints. *See* constraints
integrity rules
  in a relational database, 1–7
  See also data, integrity of; constraints
INTERNAL, 2–2
  audit records not generated by, 19–4
internal errors, tracked in trace files, 9–14

INVALID status, 16–2
  See also name of object to which status applies
IS NULL predicate, 5–7
ISO SQL standard, 1–3
  See also ANSI SQL standard
isolation levels
  choosing, 10–13
  comparing, 10–12
  read committed, 10–9
  serializable, 10–9
  setting, 10–8
ITSEC security standard, 1–3

J

jobs, 9–3
join views, 5–12
joins
  cartesian products, 13–11
  clusters and, 5–25
  cross, 13–11
  defined, 13–11
  encapsulated in views, 1–9, 5–10
  equijoins, 13–11
  nonequijoins, 13–11
  outer, 13–11

K

key values, 1–30
  See also keys
keys
  cluster, 1–12, 5–25
    See also clusters, keys
  foreign, 7–11
  hash, 5–30
  in constraints, 1–29
  indexes and, 5–20
  maximum storage for values, 5–20
  parent, 7–12
  primary, 7–10 to 7–11
  referenced, 1–30, 7–12
keys (*continued*)
  unique
    composite, 7–8, 7–10
    defined, 7–8
    how enforced, 7–8 to 7–10

L

latches
  defined, 10–19
  described, 10–27
  LRU, 9–8
LCKn. *See* lock processes (LCKn)
leaf blocks. *See* indexes
least recently used algorithm (LRU)
  database buffers and, 9–17
  latches, 9–8
LGWR. *See* log writer process
library caches
  memory allocation for, 9–22
  size of, 9–22
licensing
  concurrent usage, 17–12
  named user, 17–14
  viewing current limits, 17–13
listener processes, 9–13
local databases, 1–46
  See also see also databases, distributed
location transparency, 1–46
  See also databases, distributed
lock processes (LCKn), 1–22, 9–12
  See also Parallel Server
locks
  after committing transactions, 12–5
  automatic, 1–32, 10–16, 10–18
  conversion, 10–17
  data, 10–19 to 10–25
    duration of, 10–17
  deadlocks, 10–4, 10–17
    avoiding, 10–18

locks (*continued*)
- dictionary, 10–19, 10–26
  - clusters and, 10–27
  - duration of, 10–27
  - exclusive, 10–26
  - share, 10–26
- dictionary cache, 10–28
- distributed, 10–19
- DML acquired, 10–25
  - diagrammed, 10–24
- escalation, 10–17
- escalation does not occur, 10–5
- exclusive, 10–4
- exclusive table locks (X), 10–23
- file management locks, 10–28
- how Oracle uses, 10–16
- internal, 10–19, 10–27
- latches and, 10–27
- log management locks, 10–28
- manual, 1–33, 10–29 to 10–36
  - examples of behavior, 10–29 to 10–36
- Oracle Lock Management Services, 10–36
- overview of, 1–32, 10–3 to 10–5
- parallel cache management (PCM), 10–19
- parse, 10–27
- rollback segment, 10–28
- row (TX), 10–20
- row exclusive locks (RX), 10–22
- row share table locks (RS), 10–21
- share, 10–4
- share row exclusive locks (SRX), 10–23
- share table locks (S), 10–22
- share–sub–exclusive locks (SSX), 10–23
- sub–exclusive table locks (SX), 10–22
- sub–share table locks (SS), 10–21
- table, modes of, 10–20 to 10–22
- table (TM), 10–20
- tablespace, 10–28
- types, 10–19
- types of, 10–19

log management locks, 10–28

log sequence numbers, 1–42
- after datafile recovery, 24–14
- control files and, 22–9, 24–12
- datafiles and, 24–12
- during recovery, 24–12
- multiplexed redo logs and, 22–13
- resetting to 1, 24–16
- *See also* redo log files

log switches
- described, 22–9
- log sequence numbers, 22–9
- multiplexed redo log files and, 22–13

log writer process (LGWR)
- archiver process (ARCH) and, 22–17
- checkpoint process (CKPT) and, 9–11
- defined, 1–21
- group commits, 9–10
- manual archiving and, 22–20
- multiplexed redo log files and, 22–13
- online redo logs available for use, 22–7
- overview, 9–9 to 9–11
- redo log buffers and, 9–19
- system change numbers, 12–5
- trace files and, 22–14
- writing to online redo log files, 22–7

LOG_BUFFER parameter
- defined, 9–19
- System Global Area size and, 9–26

LOG_CHECKPOINT_INTERVAL parameter, 22–10

LOG_CHECKPOINT_TIMEOUT parameter, 22–10

LOG_FILES parameter, 9–28

logical blocks. *See* data blocks

logical database structure. *See* databases; logical structure of

logical reads limit, 17–10

logical structures. *See* structures, logical

LONG datatype
- automatically the last column, 5–6
- defined, 6–7
- restrictions on, 6–7
- storage of, 5–6

LONG RAW datatype
- defined, 6–8
- indexing prohibited on, 6–8

restrictions on, 6–7
similarity to LONG datatype, 6–8
LOW_VALUE column, of USER_TAB_COLUMNS view, 13–18
LRU. *See* least recently used algorithm

M

MAC. *See* mandatory access control
mandatory access control, 1–39
manual locking, 10–29 to 10–36
match
full, 7–14
none, 7–14
partial, 7–14
See also referential integrity
media failure, described. *See* failures, media
memory
content of, 9–15 to 9–17
cursors (statement handles), 1–19
library cache, allocation for, 9–22
overview of structures in, 1–17 to 1–21
procedures can reduce usage, 14–7
processes use of, 9–3
sessions, allocation for, 9–27
SGA size and, 9–25
shared SQL areas, 9–21
allocation for, 9–22
software code areas, 9–16 to 9–18
sort areas, 9–28
SQL statements, allocation for, 9–24
structures in, 9–15
system global area (SGA), viewing size of, 9–26
System Global Area (SGA) allocation in, 9–16
virtual, 9–15
mirroring. *See* multiplexing
MLSLABEL datatype, 6–11
modes
ARCHIVELOG, 22–18
exclusive. *See* exclusive mode
NOARCHIVELOG, 22–18
parallel. *See* parallel mode
single–task, 9–31
table lock, 10–20 to 10–22
two–task, 9–32
monitoring user actions. *See* auditing
MTS_MAXSERVERS parameter, artificial deadlocks and, 9–38
multi–threaded server
artificial deadlocks in, 9–37
dedicated server contrasted with, 9–34
defined, 9–30
described, 9–34
dispatcher processes, 9–13
example of use, 9–40
memory allocation and, 9–27
processes needed for, 9–34
restricted operations in, 9–38
SQL*Net V2 requirement, 9–13
multi–threaded servers. *See* dispatcher processes; servers, multi–threaded
multi–user environments, 9–4
See also users
multiblock writes, 9–9
multiple–process systems (multi–user systems), 9–4
See also multi–user environments
multiplexing
control files, 1–43, 22–22
recovery and, 22–4
redo log files, 1–42, 22–12
See also redo log files, multiplexed
multiversion consistency model, 1–31
multiversion concurrency control, 10–7

N

name resolution, in distributed databases, 21–4
named user licensing, 17–14
National Language Support (NLS)
character sets for, 6–4
Check constraints and, 7–16
clients and servers may diverge, 21–2
defined, 2–7
views and, 5–11

networks
  client/server architecture use of, 20–2
  communication protocols of, 20–5
  communications software for, 9–42
  dispatcher processes and, 9–13
    protocols, use of, 9—34
  distributed databases' use of, 21–2
  drivers, 9–42
  failures of, 22–3
  independence from in distributed databases, 21–8
  listener processes of, 9–13
  network authentication service, 17–4
  SQL*Net. *See* SQL*Net
  two–task mode and, 9–32, 9–33
  using Oracle on, 1–6, 1–48
NLS. *See* National Language Support
NLS_DATE_FORMAT parameter, 6–6
NOARCHIVELOG mode
  defined, 22–18
  full backups required for recovery in, 23–2
  overview, 1–43
NOAUDIT command, locks, 10–27
nodes, of distributed databases, 1–46
non–repeatable reads, 10–3, 10–13
non–unique indexes, 5–19
nonequijoins, defined, 13–11
NOT NULL constraints
  defined, 7–7
  implied in PRIMARY KEY constraints, 7–11
  UNIQUE keys and, 7–9
nulls
  as default values, 5–7
  column order and, 5–6
  converting to values, 5–7
  defined, 5–6
  equality of in UNIQUE key constraints, 7–9
  foreign keys and, 7–14 to 7–16
  how stored, 5–7
  indexes and, 5–7
  prohibited in primary keys, 7–10
  prohibiting, 7–7
  UNIQUE key constraints and, 7–9
  unknown in comparisons, 5–7
NUM_DISTINCT column, USER_TAB_COLUMNS view, 13–18
NUM_ROWS column, USER_TABLES view, 13–18
NUMBER datatype, 6–4
  internal format of, 6–5
  rounding, 6–5
NVL function, 5–7

O

object privileges, 18–4
objects
  auditing access to, overview, 1–39
  dependencies of, foreign keys, 7–12
  in a database schema. *See* schema objects
OCI. *See* Oracle call interface
offline backups
  See also backups
  consistent, 23–5
  datafiles, 23–4
  full, 23–2
  fuzzy data and, 23–5
offline redo log files. *See* redo log files, archived
online backups
  See also backups
  datafiles, 23–4
online redo log. *See* redo log files, online
OPEN_CURSORS parameter
  defined, 9–25
  managing private SQL areas, 9–22
OPEN_LINKS parameter, 9–28
operating systems
  authentication by, 17–3
  block size, 3–3
  privileges required for CONNECT INTERNAL, 2–2
  roles and, 18–14
operating–system communications software, 9–42
operations, in a relational database, 1–7
OPTIMAL storage parameter, 3–22
  See also parameters

optimization
  cost–based
    choosing an access path, 13–15
    examples of, 13–16
    remote databases and, 13–17
  described, 13–2
  rule–based
    choosing an access path, 13–14
    examples of, 13–14
  selectivity of queries and, 13–16
Oracle
  adherence to industry standards, 1–3 to 1–5, 7–5, 11–2
  architecture, 1–7, 1–17 to 1–25
  client/server architecture of, 20–2
  compatibility, 1–4
  configurations of, 9–29 to 9–38
Oracle (*continued*)
  connectibility of, accessing non–Oracle databases, 21–8
  connectibility, 1–4
  data access, 1–24
  examples of operations, 1–23, 9–39
  features, 1–2, 1–4
  instances, defined, 1–19
  licensing of, 17–12
  Oracle Server, 1–4
  Parallel Server option, 1–6
  portability, 1–4
  processes of, 1–20, 9–5
  recovery features of, 24–2
  scalability of, 20–4
  single–process Oracle, 9–3
  SNMP support, 1–3
  Trusted. *See* Trusted Oracle
  using on networks, 1–4
    See also networks; SQL*Net
Oracle blocks. *See* data blocks
Oracle call interface (OCI), 9–41
Oracle code, 9–41
Oracle Forms
  application triggers contrasted with database triggers, 15–5
  object dependencies and, 16–10
  Version 3, 11–8
Oracle Parallel Server, isolation levels, 10–11
Oracle program interface (OPI), 9–41
  See also program interface
Oracle Server. *See* databases
Orange Book security standard, 1–3
OS_AUTHENT_PREFIX parameter, operating system authentication and, 17–4
outer joins, defined, 13–11

P

packages
  as program units, 1–28
  auditing, 19–7
  bodies of, 14–9, 14–13
  cached in the shared pool, 14–14
  compilation and entry into database of, 14–14
packages (*continued*)
  defined, 14–3 to 14–5
  dependency tracking in, 14–14
  encapsulate procedures, 14–12
  errors in, 14–15
  examples of, 14–9 to 14–13, 18–8, 18–9
  execution of, 14–15
  execution steps of, 14–16
  for locking, 10–36
  introduced, 14–2
  overview of, 1–11
  performance effect of, 14–13
  privilege to execute, 18–7
  privileges
    divided by construct, 18–8
    executing, 18–8
  public and private data and procedures, 14–12
  session state and, 16–8
  shared SQL areas and, 9–23
  specifications of, 14–9, 14–13
  stand–alone procedures contrasted with, 14–8
  uses of, 14–12
  validation status, 14–16
  verifying user access, 14–15
pages. *See* data blocks
parallel mode, 2–4

parallel query option, space management, 3–11
parallel recovery, 24–5
  See also recovery
Parallel Server
  checkpoints and, 22–11
  concurrency limits and, 17–13
  databases and instances, 9–2
  distributed locks, 10–19
  exclusive mode, rollback segments and, 3–24
  file and log management locks, 10–28
  introduced, 1–6
  lock processes, 9–12
  modes of, 2–4
  mounting a database using, 2–4
  named user licensing and, 17–14
  PCM locks, 10–19
  shared mode, rollback segments and, 3–24
  system change numbers, 9–10
  system monitor process and, 9–11
  threads of online redo log, 22–16
  See also lock processes
parameter files
  described, 2–6 to 2–8
  example of, 2–7
  introduced, 2–3
parameters
  initialization, locking behavior, 10–18
  national language support, 2–7
  ROLLBACK_SEGMENTS, 3–23
  SORT_AREA_SIZE, 3–28
  TRANSACTIONS, 3–23
  TRANSACTIONS_PER_ROLLBACK_SEGMENT, 3–23
  See also names of individual parameters not listed; parameter files
parent tables, 7–12
parse locks, 10–27
parse trees
  construction of, 11–6
  location of, 9–21
  of procedures, 14–14
parsing
  parse calls, 11–7
  performed, 11–7
partial backups, 23–3
  See also backups
partition views, 5–13
passwords
  connecting without, 17–3
  database user authentication and, 17–4
  encryption, 17–4
  password files, 17–5
  used in roles, 1–37
pcode, 15–14
PCTFREE storage parameter
  how it works, 3–5 to 3–7
  PCTUSED and, 3–7
PCTUSED storage parameter
  how it works, 3–6 to 3–9
  PCTFREE and, 3–7
performance
  checkpoints effect, 9–11 to 9–13, 22–9
  clusters and, 5–25
  constraint effects on, 7–6
  distributed databases and, 21–8
  group commits, 9–10
  packages and, 14–13
  parallel recovery and, 24–5
  procedures and, 14–7
  procedures effect on, 14–8
  resource limits and, 17–8
  SGA size and, 9–25
  structures that improve, 1–11 to 1–14
  viewing execution plans, 13–5
persistent areas, 9–21
PGA. *See* Program Global Area
phantom reads, 10–3
phantoms, 10–13
physical database structure. *See* databases; physical structure of
PL/SQL
  and database triggers, 1–30
  anonymous blocks, 11–7, 14–8
  auditing of statements within, 19–4
  blocks, 11–7
  engine, products with, 11–8
  exception handling, 11–9
  executing, 11–8
  introduced, 1–10
  language constructs, 11–9
  overview, 11–7 to 11–11
  overview of, 1–27, 14–5 to 14–8
  parse locks, 10–27
  PL/SQL blocks, 11–7

PL/SQL (*continued*)
  PL/SQL engine, 11–8
  procedures and, 14–5
  program units, shared SQL areas and, 9–23
  stored procedures, 11–7
  trigger actions, 15–7
PMON. *See* process monitor process
portability, 1–4
prefixes, data dictionary view, 8–5
PRIMARY KEY constraints
  described, 7–10
  indexes used to enforce, 7–11
    name of, 7–11
  maximum number of columns in, 7–11
  NOT NULL constraints implicit in, 7–11
primary keys
  described, 7–10 to 7–12
  integrity constraints and. *See* data, integrity of
private, rollback segments. *See* rollback segments, private
private SQL areas
  cursors and, 9–25
  described, 9–20
  how managed, 9–22
  persistent areas, 9–21
  runtime areas, 9–21
privileges
  auditing the use of, overview of, 1–38
  auditing use of, 19–7
  granting, 1–36, 18–3
    examples of, 18–8, 18–9
    to roles, 18–11
  grouping into roles, 1–36
  object, 18–3
    granting and revoking, 18–4
    overview of, 1–36
  overview of, 1–35, 18–2 to 18–6
  procedures, 18–7
    altering, 18–7
    creating, 18–7
    divided by construct, 18–8
    executed under owner's, 18–7
    executing, 18–7
  revoked, object dependencies and, 16–6
  roles, restrictions on, 18–13
privileges (*continued*)
  roles and, 18–10
  system, 18–2
    granting, 18–3
    overview of, 1–36
  to start up or shut down a database, 2–2
  trigger, executed under owner's, 18–8
  views, 18–5
    creating, 18–5
procedures
  anonymous PL/SQL blocks contrasted with, 14–8
  applications and, 14–8
  applications for, 14–7
  auditing, 19–7
  cached in the shared pool, 14–14
  compilation and entry into database of, 14–14
  contrasted with functions, 1–28
  cursors and, 11–9
  dependency tracking in, 14–8, 16–5
  described, 14–5
  design and use of, 14–6
  encapsulating in packages, 14–12
  errors in, 14–15
  examples of, 18–8, 18–9
  execution of, 14–15
  execution steps of, 14–16
  functions contrasted with, 14–6
  introduced, 14–2
  INVALID status, 16–2, 16–5
  memory usage and, 14–7
  packages can contain, 14–3
  parse trees produced for, 14–14
  performance effect of, 14–7, 14–8
  PL/SQL and, 11–7
  prerequisites for compilation of, 16–4
  privileges
    executed under owner's, 18–7
    to execute, 18–7
  privileges on
    create or alter, 18–7
    executing, 18–7
    executing any, 18–7
  pseudocode (P code), 14–15
  security and, 14–7
  security enhanced by, 18–7

procedures (*continued*)
shared SQL areas and, 9–23
stand–alone, 14–8
stored procedures, 11–7
triggers contrasted with, 15–2
VALID status, 16–3
validation status, 14–16
verifying user access, 14–15

Process Global Area (PGA)
allocation of, 9–26
See also Program Global Area (PGA)

process monitor process (PMON)
cleans up timed–out sessions, 17–10
described, 1–22, 9–11
detects failures, 1–40

processes
archiver (ARCH), 1–22, 9–12
background, 1–21, 9–5
diagrammed, 9–6
checkpoint (CKPT), 1–21, 9–11
checkpoints and, 9–8
database writer (DBWR), 1–21, 9–7
dedicated server, 9–37
described, 9–2
dispatcher (Dnnn), 1–22, 9–13
distributed transaction resolution and, 9–12
during recovery, 24–5
failure in, 22–3
listener, 9–13
shared servers and, 9–34
lock (LCKn), 1–22, 9–12
log writer (LGWR), 1–21, 9–9
multi–threaded
artificial deadlocks and, 9–37
client requests and, 9–37
multi–threaded server requires, 9–34
multiple–process Oracle, 9–4
Oracle, 1–20, 9–5
overview of, 1–20
process monitor (PMON), 1–22, 9–11
recoverer (RECO), 1–22, 9–12
and in–doubt transactions, 1–47
server, 1–20, 1–45, 9–5
dedicated, 9–32
shared, 9–13
shadow, 9–32
shared server, 9–37
single–process Oracle, 9–3

processes (*continued*)
snapshot refresh (SNPn), 9–13
structure, 9–3
system monitor (SMON), 1–21, 9–11
trace files for, 9–14
user, 1–20, 9–5
allocate PGAs, 9–26
recovery from failure of, 9–11
sharing server processes, 9–13

processing, distributed, 1–45 to 1–48

profiles
overview of, 1–38
when to use, 17–11

Program Global Area (PGA), 9–26
contents of, 9–26
defined, 1–20
multi–threaded servers, 9–37
non–shared and writable, 9–27
session information and, 9–27
size of, 9–28

program interface
Oracle side (OPI), 9–41
overview of, 1–22, 9–41 to 9–42
single–task mode in, 9–31
structure of, 9–41
two–task mode in, 9–33
user side (UPI), 9–41

program units
overview of, 1–10
PL/SQL and, 14–5
prerequisites for compilation of, 16–4

pseudocode (P code) for procedures, 14–15

pseudocolumns, 6–9
ROWID, 6–9

public, rollback segments. *See* rollback segments, public

PUBLIC user group, 17–7
difference from public variables, 14–13

Q

queries
and partition views, 5–13
compound, defined, 13–12
default locking of, 10–24 to 10–26
defined, 13–11

queries (*continued*)
distributed, 21–6
distributed or remote, 21–6
in DML. *See* Data manipulation statements (DML)
location transparency and, 21–7
merged with view queries, 5–11
non–Oracle databases and, 21–8
phases of, 10–6
read consistency of. *See* read consistency
selectivity of, 13–17
stored, 1–9
stored as views, 5–8
See also views
triggers' use of, 15–13
query servers, extent allocation, 3–11
quotas
revoking tablespace access and, 17–7
setting to zero, 17–7
tablespace, 1–37, 17–6
tablespaces, temporary segments ignore, 17–7

R

RAW datatype, 6–8
RDBMS, processes of. *See* Oracle, processes of
read committed
isolation, 10–9
vs. serializable, 10–12
read consistency
defined, 1–31
multiversion consistency model, 1–31
rollback segments and, 3–17 to 3–19
snapshot too old message, 10–6
transaction level, 10–7 to 10–9
transactions, 1–31 to 1–33
triggers and, 15–13 to 15–15
read snapshot time, 10–13
read uncommitted, 10–3
read–only tablespaces
See also tablespaces, read–only
backing up, 23–8 to 23–10
described, 4–8 to 4–9
restrictions on, 4–9 to 4–10
read–only transactions. *See* transactions, read–only
readers block writers, 10–13
reads
data block, limits on, 17–10
dirty, 10–3
repeatable, 10–7
RECO. *See* recoverer process
recoverer process (RECO), 1–22
and in–doubt transactions, 1–47
overview of, 9–12 to 9–14
recovery
basic steps, 1–44 to 1–46
closed database, 24–10
control files during incomplete, 24–16
database, overview of, 1–40 to 1–42
database buffers and, 24–3
diagrammed, 24–6
distributed processing in, 9–12
features of, 24–2
full backups and, 23–2
in parallel, 24–5
incomplete, 24–14
requirements of, 24–16
instance, 24–7
automatic, 2–4
required after abort, 2–5
system monitor process (SMON) and, 9–11
media
ARCHIVELOG mode, 24–8
backup control files used, 24–11
backup datafiles and, 24–11
cancel–based, 24–15
complete, 24–10
control files and, 22–17
datafile, 24–10
disabled, 22–18
dispatcher processes and, 9–38
enabled, 22–18
examples of, 24–11 to 24–15
incomplete, 24–14, 24–16
log sequence numbers during, 24–12
mechanisms involved, 24–11 to 24–15
NOARCHIVELOG mode, 24–8
open database–offline tablespace, 24–10

recovery
  media (*continued*)
    overview of, 24–8 to 24–17
    read–only tablespaces and, 24–8
    redo log files applied, 24–12
    rolling back, 24–13
    time–based, 24–15
  of distributed transactions, 2–5
  overview of, 24–3 to 24–5
  process, 9–11, 22–3
  recommendations for, 24–6
  requirements of, 24–2
  rolling back during, 24–4
  rolling forward and, 24–3
  statement failure, 22–2
  steps of, 24–4
  structures used in, 1–42, 22–5
recursive SQL, cursors and, 9–25
redo entries, content of, 22–6 to 22–8
redo log buffers
  circularity, 9–10
  committing a transaction, 9–10
  log writer process and, 9–19
  overview, 9–19
  size of, 9–19
  writing, 9–9
  writing of, 22–7
redo log files
  ”fuzzy” data in backups and, 23–5
  active (current), 22–8
  applied during media recovery, 24–12
  archived, 1–43
    advantages of, 22–16
    automatic, 22–19
    contents of, 22–17
    control files and, 22–17
    errors in archiving, 22–20
    log switches and, 22–9
    manually, 22–20
    mechanics of archiving, 22–16
  archiver process (ARCH) and, 9–12
  available for use, 22–7
  buffer management, 9–9
  contents of, 22–6 to 22–8
  distributed transaction information in, 22–7
  files named in control file, 22–21
  groups, 22–13
redo log files (*continued*)
  inactive, 22–8
  log sequence numbers of
    defined, 1–42
    uses, 22–9
  log switches, 22–9
  log writer process, 9–9, 22–7
  members, 22–13
  mirrored
    archiver process (ARCH) and, 22–17
    if all inaccessible, 22–13
    if some inaccessible, 22–13
    log switches and, 22–13
  mode of, 1–43
  multiplexed, 1–42, 22–12
    diagrammed, 22–13
    purpose of, 1–15
  offline. *See* redo log files, archived
  online, 1–42, 22–6
    after checkpoint failure, 22–11
    loss of, 24–14
    recovery use of, 22–6 to 22–9
    requirement of two, 22–7
    threads of, 22–16
  overview of, 1–15, 1–42
  parallel recovery, 24–5
  physical database structure, 1–5
  recovery and, 22–5 to 22–7
  resetting log sequence numbers, 24–16
  rollback segments and, 3–17
  rolling forward and, 24–3
  when temporary segments in, 3–29
  written before transaction commit, 9–10
  See also recovery
referenced
  keys, 1–30, 7–12
  objects, 16–2
  tables, 7–12
REFERENCES privilege, when granted
    through a role, 18–13
referential integrity
  cascade rule, 7–3
  distributed databases and, 7–4
  examples of, 7–17 to 7–20
  intermediate states and, 7–18
  partially null foreign keys, 7–14
  PRIMARY KEY constraints, 7–10

referential integrity (*continued*)
restrict rule, 7–3
self–referential constraints, 7–13, 7–17
set to default rule, 7–3
set to null rule, 7–3
See also foreign keys
refresh, snapshot refresh process, 9–13
relational database management systems (RDBMS's)
principles, 1–7 to 1–9
See also database management systems
SQL and, 11–2
relations. *See* tables
remote databases, 1–46
See also see also databases, distributed
remote transactions, 21–6
repeatable reads, 10–3
replicating data, 1–47, 21–10
replication option, 1–47, 21–10
reserved words, 11–2
resource limits, 17–8
call level, 17–9
connect time per session, 17–10
CPU time limit, 17–9
determining values for, 17–11
idle time per session, 17–10
logical reads limit, 17–10
overview of, 1–38
private SGA space per session, 17–11
session level, 17–9
sessions per user, 17–10
RESOURCE role, 18–14
response queues, 9–37
See also dispatcher processes (Dnnn)
restricted mode, starting instances in, 2–3
REVOKE command, locks, 10–27
roles
application, 18–11
CONNECT role, 18–14
DBA role, 18–14
DDL statements and, 18–13
dependency management in, 18–13
disabling, 18–11
enabling, 18–11
EXP_FULL_DATABASE, 18–14
functionality, 18–2, 18–11
roles (*continued*)
granting, 18–3, 18–12
to other roles, 18–11
users capable of, 18–12
IMP_FULL_DATABASE, 18–14
in applications, 1–37
management using the operating system, 18–14
naming, 18–12
overview of, 1–36
predefined, 18–14
RESOURCE role, 18–14
restrictions on privileges of, 18–13
revoking, 18–12
users capable of, 18–12
schemas do not contain, 18–12
security domains of, 18–12
use of passwords with, 1–37
user, 18–11
uses of, 18–10 to 18–12
rollback
See also rollback segments; transactions
caused by PMON, 1–40
defined, 1–25
described, 12–6
during recovery, 1–44
rollback entries, 3–17
See also rollback segments
rollback segments
access to, 3–17
acquired during startup, 2–5
allocating new extents for, 3–21
allocation of extents for, 3–19
clashes when acquiring, 3–23
committing transactions and, 3–18
contention for, 3–19
deallocating extents from, 3–22
deferred, 3–27
defined, 1–14
dropping, 3–23
restrictions on, 3–27
how transactions write to, 3–19
in–doubt distributed transactions and, 3–21
invalid, 3–25
locks on, 10–28
media recovery use of, 24–13
moving to the next extent of, 3–19
number of transactions per, 3–19

rollback segments (*continued*)
offline, 3–25, 3–26
offline tablespaces and, 3–27
online, 3–25, 3–26
overview of, 3–16
parallel recovery, 24–4
partly available, 3–25 to 3–28
private, 3–23
public, 3–23
read consistency and, 1–31 to 1–33, 3–17, 10–5
recovery needed for, 3–25 to 3–28
states of, 3–24 to 3–27
viewing, 3–27
SYSTEM rollback segment, 3–24
transactions and, 3–18
use of in recovery, 1–43
when acquired, 3–23
when used, 3–17
written circularly, 3–18
See also segments; rollback
rolling back during recovery, 24–4, 24–13
See also recovery; redo log files
rolling back transactions, 12–2, 12–6
See also transactions, rolling back
rolling forward, during recovery, 1–44
rolling forward during recovery, 24–3, 24–12
See also recovery; redo log files
root blocks, 5–32 to 5–34
row cache, 9–23
row data (section of data block), 3–4
row directories, 3–4, 5–6
See also rows
row locking, and serializable transactions, 10–11
ROW LOCKING parameter, 10–29
row pieces, 5–4 to 5–7
headers, 5–5
how identified, 5–6
row sources, 13–3 to 13–5
row triggers, 15–7
when fired, 15–11
row–level locking, 10–13
ROWID datatype, 6–9
ROWIDs, 5–6
accessing, 6–9
changes in, 6–9
examples of use, 6–10
in non–Oracle databases, 6–10
internal use of, 6–10
of clustered rows, 5–6
retained during migration, 3–10
sorting indexes by, 5–22
table access by, 13–13
ROWLABEL pseudocolumn, 6–12
rows
addresses of, 5–6
chaining across blocks, 3–10, 5–4
clustered, 5–6
ROWIDs of, 5–6
defined, 1–9
described, 5–3
directories in, bytes per row, 5–6
format of in data blocks, 3–4 to 3–6
headers. *See* row pieces, headers
locking, 10–20
locks on, 10–21 to 10–23
migration between data blocks, 3–10
pieces of. *See* row pieces
row sources, 13–3 to 13–5
ROWIDs used to locate, 13–13
shown in ROWIDs, 6–9
size of, 5–4
storage format of, 5–4
triggers on, 15–7
when ROWID changes, 6–9
rule–based optimization, 13–6
runtime areas, 9–21

S

same–row writers block writers, 10–13
savepoints
described, 12–6
overview of, 1–27
rolling back to, 12–6
See also transactions
scaling your database, 20–4

scans, 13–13
schema objects
  allocated segments, 4–3
  auditing, 19–7
  creating, tablespace quota required, 17–6
  default tablespace for, 17–6
  defined, 1–5
  dependencies between, 16–2
  dependencies of
    and distributed databases, 16–10
    and views, 5–12
    on non–existence of other objects, 16–6
    package tracking of, 14–14
    procedure tracking of, 14–8
    triggers manage, 15–11
  dependent on lost privileges, 16–6
  distributed database naming conventions for, 21–4
  global names, 21–4
schema objects (*continued*)
  in a revoked tablespace, 17–7
  information about, 8–2
  INVALID status, 16–2
  overview of, 1–9, 5–2
  privileges on, 18–3
    DDL, 18–5
  relationship to datafiles, 4–10, 5–2
  trigger dependencies on, 15–15
  VALID status, 16–3
schemas
  associated with users, 1–33, 5–2
  contents of, 5–2
  contrasted with tablespaces, 5–2
  defined, 17–2
  objects in, 5–2
    See also schema objects
  See also users
SCN. *See* system change numbers (SCN)
security
  application enforcement of, 1–37
  auditing, 19–2, 19–6
  auditing user actions. *See* auditing
  data, 1–34
  described, 1–33
  discretionary, 17–1
  discretionary access control, 1–34
  domains, 1–35
security (*continued*)
  enforcement mechanisms, 1–34
  procedure and package access validation, 14–15
  procedures and, 14–7
  procedures enhance, 18–7
  program interface enforcement of, 9–41
  system, 1–33
  views and, 5–10
  views enhance, 18–6
  See also privileges, roles
security domains
  enabled roles and, 18–11
  tablespace quotas, 17–6
segments
  allocating, temporary, 3–28
  allocating extents for, 3–10 to 3–12
  data, 3–16
  deallocating extents from, 3–14
  defined, 3–3
  header block, 3–11
  index, 3–16
  minimum number of extents in, 3–10
  overview, 1–14
  overview of, 3–15 to 3–17
  rollback, 3–16
  temporary, 1–14, 3–28
    cleaned up by SMON, 9–11
    dropping, 3–15
    ignore quotas, 17–7
    operations that require, 3–28
    tablespace containing, 3–28
  used to store schema objects, 4–3
  See also data segments; index segments; rollback segments; temporary Segments
selectivity of queries, 13–17
sequences, 1–10, 5–16
  auditing, 19–7
  independence from tables, 5–16
  length of numbers, 5–16
  number generation, 5–16
  VALID status, 16–3
serializable
  isolation, 10–9
  vs. read committed, 10–12
SERIALIZABLE parameter, 10–29
  locking, 10–18

server processes. *See* processes
servers
  client/server architecture, 20–2
  dedicated, 1–20
    multi–threaded contrasted with, 9–34
  dedicated processes, 9–32
  dedicated server architectures, 9–30
  defined, 1–45
  multi–threaded, 1–20
    dedicated contrasted with, 9–34
    processes of, 9–37
  multi–threaded server architectures, 9–30
  processes of, 1–20
  shared. *See* servers, multithreaded
session control statements, 1–25, 11–5
sessions
  connections contrasted with, 9–30
  defined, 9–30
  limit on concurrent, 1–38
    by license, 17–12
    imposed by DBA, 17–13
  limits per user, 17–10
  package state and, 16–8
  resource limits and, 17–9
  time limits on, 17–10
  when auditing options take effect, 19–5
SET TRANSACTION READ ONLY statement, 3–18
  See also transactions
sets, LRU latches, 9–8
share locks
  DDL locks, 10–26
  defined, 10–4
  share table locks (S), 10–22
Shared Global Area (SGA). *See* System Global Area (SGA)
shared mode. *See* Parallel Server, shared mode
shared pool
  allocation of, 9–23
  ANALYZE command and, 9–24
  anonymous PL/SQL blocks and, 14–8
  compiled PL/SQL code in, 14–14
  dependency management and, 9–24
  described, 9–20
  flushing, 9–25
  object dependencies and, 16–7
  overview of, 1–19
shared pool (*continued*)
  procedures and, 14–7
  row cache and, 9–23
  size of, 9–20
shared servers
  cannot CONNECT INTERNAL to, 2–2
  See also servers, multi–threaded
shared SQL areas
  ANALYZE command and, 9–24
  dependency management and, 9–24
  described, 9–20, 11–6
  how managed, 9–22
  memory allocation for, 9–22
  overview of, 1–19
  parse locks and, 10–27
  procedures, packages, triggers and, 9–23
  size of, 9–21
SHARED_POOL_SIZE parameter, 9–22
  System Global Area size and, 9–26
shutdown, 2–6
  abnormal, 2–3
  checkpoints and, 22–10
  deallocation of the SGA, 9–16
  described, 2–2
  prohibited by dispatcher processes, 9–38
  steps, 2–5
SHUTDOWN ABORT, 2–6
single–process systems (single–user systems), 9–3
single–task mode, 9–31
site autonomy, 1–46, 21–3
  See also databases, distributed
SMALL_TABLE_THRESHOLD parameter, 9–18
SMON. *See* system monitor process
snapshot refresh process (SNPn), 9–13
snapshot too old message, 10–6
snapshots, 1–47, 21–10
  refreshing, 9–13
SNMP support, 1–3
SNPn. *See* snapshot refresh process
software code areas, 9–16 to 9–18
  shared by programs and utilities, 9–16
sort areas, 9–28
sort direct writes feature, 9–29

SORT_AREA_RETAINED_SIZE parameter, 9–28
SORT_AREA_SIZE parameter, 3–28, 9–28
space management
  compression of free space, 3–9
  extent management, 3–10 to 3–15
  parallel query option, 3–11
  PCTFREE, 3–5
  PCTUSED, 3–6
  row chaining, 3–10
  segments, 3–15 to 3–28
SQL
  cursors used in, 11–6
  Data Definition Language (DDL), 11–4
  embedded in other languages, 1–25
  embedded SQL, 11–5
  functions, 11–2
  overview of, 1–24, 11–2 to 11–7
  parsing of, 11–6
  PL/SQL. *See* PL/SQL
  PL/SQL and, 11–7, 14–5
  recursive, 11–6
    cursors and, 9–25
  reserved words, 11–2
  session control statements, 11–5
  shared areas. *See* Shared SQL
  shared SQL, 11–6
  statement–level rollback, 12–4
  statements, memory allocation for, 9–24
  statements in, 11–2
  system control statements, 11–5
  transaction control statements, 11–4
  transactions, 12–2
  transactions and, 1–25
  types of statements in, 1–24 to 1–26, 11–3
  See also transactions
SQL areas
  private, 9–20 to 9–22
    persistent, 9–21
    runtime, 9–21
  shared, 9–20 to 9–22
SQL statements
  audit records of, when generated, 19–4
  auditing, 19–6, 19–9
  complex, defined, 13–12
  dictionary cache locks and, 10–28
SQL statements (*continued*)
  distributed
    defined, 13–12
    optimization of, 13–18 to 13–20
  distributed databases and, 21–6
  execution plans of, 13–2
  failure in, 22–2
  in trigger actions, 15–7
  number of triggers fired by single, 15–11
  optimization of, 13–10 to 13–12
  parse locks, 10–27
  privileges required for, 18–3
  procedural extensions to, 14–5
  referencing dependent objects, 16–4
  resource limits and, 17–9
  successful execution, 12–3
  trigger events and, 15–6
  triggers fired by, 15–3
  triggers on, 15–8
SQL*Connect, 21–8
SQL*Net, 1–6
  applications and, 20–5
  client/server systems use of, 20–5
  how it works, 20–5
  multi–threaded server V2 requirement, 9–13
  network independence and, 21–8
  overview of, 1–48, 20–5 to 20–6
  program interface and, 9–42
SQL*Plus, 11–9
SQL_TRACE parameter, 9–14
SQL92, 10–2
standards
  Oracle adherence, 1–3 to 1–5
  Oracle adherence with, 7–5
  See also names of particular standards
standby database, 22–23
startup
  allocation of the SGA, 9–16
  described, 2–2
  exclusive mode, 2–4
  fast warmstart, 24–7
  forcing, 2–3
  parallel mode, 2–4
  prohibited by dispatcher processes, 9–38
  recovery during, 24–7

startup (*continued*)
  restricted mode, 2–3
  steps, 2–3
statement triggers
  described, 15–8
  when fired, 15–11
statement–level read consistency, 10–7
statements
  auditing the use of, overview, 1–38
  handles. *See* cursors
  in SQL
    overview, 1—24
    types of, 1—24 to 1—26
statistics
  caches, 9–11
  checkpoint, 9–11
  optimizer use of, 13–6 to 13–8
storage
  datafiles, 4–10
    See also datafiles
  logical structures, 4–3, 5–2
  of hash clusters, 5–28
  of index clusters, 5–26
  of indexes, 5–20
  of nulls, 5–7
  of views, 5–11
  procedures and packages in the database, 14–14
  restricting for users, 17–6
  revoking tablespaces and, 17–7
  tablespace quotas and, 17–7
  triggers, 15–2, 15–14
  user quotas on, 1–37
STORAGE clause, using, 3–15
storage parameters
  OPTIMAL (in rollback segments), 3–22
  setting, 3–15
stored functions. *See* functions
stored procedures
  SYSTEM tablespace and, 14–15
  See also procedures
structure, of databases
  logical, 1–5
  See also databases
Structured Query Language. *See* SQL
structures, 1–7
  databases, physical, 1–5
  in a relational database, 1–7
  locking, 10–26
  logical, 1–8
  physical, 1–15
  schema objects, 5–2
    See also schema objects
subqueries
  Check constraints prohibit, 7–16
  in remote updates, 21–6
survivability, 22–23
synonyms
  constraints indirectly affect, 7–5
  described, 5–17 to 5–18
  for data dictionary views, 8–5
  inherit privileges from object, 18–3
  overview of, 1–11
  private, 5–17
  public, 5–17
  uses of, 5–17
  VALID status, 16–3
  See also dependencies
SYS username
  audit records not generated by, 19–4
  data dictionary tables owned by, 8–3
  security domain of, 17–2
SYS.AUD$ view, purging, 8–4
SYSDBA privilege, 2–2
SYSOPER privilege, 2–2
SYSTEM, security domain of, 17–2
system change numbers (SCN)
  change–based recovery, 24–15
  committed transactions, 12–5
  defined, 12–5
  read consistency and, 10–6, 10–7
  redo logs, 9–10
  when determined, 10–6, 22–7
system control statements, 1–25, 11–5
System Global Area (SGA)
  allocating, 2–3
  contents of, 9–17, 9–25
  data dictionary cache, 8–4
  database buffer cache, 9–17
  determining size, 2–6

System Global Area (SGA) (*continued*)
  diagram, 9–2
  fixed, 9–26
  limiting use of in multi–threaded server, 17–11
  overview of, 1–19, 9–16 to 9–17
  redo log buffer, 9–19
  shared and writable, 9–16
  shared pool, 9–20
  size of, 9–25
  when allocated, 9–16
system monitor process (SMON)
  defined, 1–21
  Parallel Server and, 9–11
system privileges
  ADMIN OPTION, 18–3
  described, 18–2
  granting, users capable of, 18–3
  revoking, users capable of, 18–3
SYSTEM rollback segment, 3–24
SYSTEM tablespace, 4–4
  See also tablespaces
  online requirement of, 4–6
  procedure storage and, 14–15

T

table directories, 3–4
tables
  affect dependent views, 16–4
  auditing, 19–7
  base, 1–9
    data dictionary use of, 8–3
    relationship to views, 5–9
  child, 7–12
  clustered, 5–23
  contain integrity constraints, 1–29
  contained in tablespaces, 5–4
  controlling space allocation for, 5–3
  dependent, 7–12
  hash, 5–32
  how data is stored in, 5–3
  indexes and, 5–18
  locks on, 10–20, 10–21, 10–23
  maximum number of columns in, 5–9
  overview of, 1–9, 5–3
  parallel query option, 3–11
tables (*continued*)
  parent, 7–12
  presented in views, 5–8
  privileges on, 18–4
  replicating, 1–47, 21–10
  replication, 1–47
  snapshots, 1–47, 21–10
  specifying tablespaces for, 5–4
  triggers used in, 15–2
  VALID status, 16–3
  virtual or viewed. *See* views
tablespaces
  backups and checkpoints, 22–10
  contrasted with schemas, 5–2
  default for object creation, 1–37, 17–6
  described, 4–3
  during incomplete recovery, 24–16
  how specified for tables, 5–4
  locks on, 10–28
  media recovery and, 24–10
  offline, 4–6 to 4–8
    and index data, 4–8
    not if read–only, 4–8
    remain offline on remount, 4–7
    what happens to datafiles, 4–11
  online, 4–6 to 4–8
    not if read–only, 4–8
    what happens to datafiles, 4–11
  online or offline, 1–8
  overview of, 1–8 to 1–10, 4–3 to 4–9
  quotas on, 1–37, 17–6 to 17–7
    limited and unlimited, 17–7
    no default, 17–6
  read–only, 4–8
    dropping objects from, 4–8
    making tablespaces read–only, 4–8
    media recovery and, 24–8
  relationship to datafiles, 4–2
  revoking access from users, 17–7
  size of, 4–6
  temporary, 1–37, 4–9
    default for user, 17–6
  used for temporary segments, 3–28
tasks, 9–3
temporary segments
  allocating, 3–28
  deallocating extents from, 3–15

temporary segments (*continued*)
dropping, 3–15
ignore quotas, 17–7
operations that require, 3–28
tablespace containing, 3–28
when not in redo log, 3–29
See also segments, temporary
temporary tablespaces, 4–9
threads, 9–3
online redo log, 22–16
three–valued logic (true, false, unknown), produced by nulls, 5–7
time–outs, 9–9
timestamps, distributed dependency checking and, 16–9
TO_DATE function, 6–6
trace files
described, 9–14
log writer process and, 22–14
transaction control statements, 1–25, 11–4
transaction set consistency, 10–13
transaction set consistent, 10–12
transaction tables, 3–17
reset at recovery, 9–11
transactions
assigning system change numbers, 12–5
assigning to rollback segments, 3–18
committing, 1–26, 9–10, 12–3, 12–5
group commits, 9–10
use of rollback segments, 3–18
writing redo log buffers and, 22–7
concurrency and, 10–17
deadlocks and, 10–4
described, 12–2
discrete transactions, 12–7
distributed
deadlocks and, 10–18
resolving automatically, 9–12
two–phase commit, 1–46
two–phase commit and, 21–6
See also transactions, in–doubt
distribution among rollback segments of, 3–19
end of, 12–4
transactions (*continued*)
in–doubt
limit rollback segment access, 3–27
resolving automatically, 1–47, 2–5
resolving manually, 1–47
rollback segments and, 3–21
use partly available segments, 3–26
manual locking of, 10–29
overview of, 1–25
read consistency of, 1–31 to 1–33, 10–7
read–only, 1–32, 10–8
not assigned to rollback segments, 3–18
redo log files written before commit, 9–10
rollback segments and, 3–18
transactions
rolling back, 1–26, 12–6
and offline tablespaces, 3–27
partially, 12–6
use of rollback segments, 3–17
savepoints in, 1–27, 12–6
serializable, 10–8
space used in data blocks for, 3–5
start of, 12–4
statement level rollback and, 12–4
system change numbers, 9–10
terminating the application and, 12–5
termination of, by PMON, 1–40
transaction control statements, 11–4
triggers and, 15–13 to 15–15
writing to rollback segments, 3–19
TRANSACTIONS parameter, 3–23
TRANSACTIONS_PER_ROLLBACK_SEGMENT parameter, 3–23
triggers
action, 15–7
timing of, 15–8
AFTER, 15–8
as program units, 1–28
auditing, 19–8
BEFORE, 15–8
can call procedures, 14–2
cascading, 15–3
constraints apply to, 15–11
constraints contrasted with, 15–5
correlation names use of, 15–7

triggers (*continued*)
  data access and, 15–13
  defined, 1–30
  dependency management of, 15–15, 16–5
  disabled, 15–11
  enabled, 15–11
  enforcing data integrity with, 7–4
  events, 15–6
  examples of, 15–10, 15–13
  firing (executing), 15–2, 15–15
    privileges required, 15–15
    steps involved, 15–11
    timing of, 15–11
  INVALID status, 16–2, 16–5
  maintain data integrity, 1–30
  object dependencies of, 15–11
  vs. Oracle Forms triggers, 15–4
  overview of, 15–2 to 15–5
  parts of, 15–5
  privileges for executing, 18–8
  procedures contrasted with, 15–2
  prohibited in views, 5–9
  restrictions, 15–7
  row, 15–7
  sequence for firing multiple, 15–11
  shared SQL areas and, 9–23
  statement, 15–8
  storage of, 15–14
  types of, 15–7
  UNKNOWN does not fire, 15–7
  uses of, 15–3 to 15–5
  when to use instead of constraints, 7–4
  See also dependencies
Trusted Oracle
  described, 1–39
  mandatory access control, 1–39
  MLS_LABEL datatype, 6–11
  mounting multiple databases in, 9–2
two–phase commit
  described, 21–6
    See also transactions, distributed
  manual override of, 1–47
two–task mode
  described, 9–32
  listener process and, 9–13
  network communication and, 9–32
  program interface in, 9–33

U

undo. *See* rollback
unique indexes, 5–19
UNIQUE key constraints, 7–8
  composite keys and nulls, 7–10
  indexes used to enforce, 7–9
  maximum number of columns in, 7–9
  NOT NULL constraints and, 7–10
  nulls and, 7–9
  size limit of, 7–9
unique keys, 1–30
  See also data, integrity of
updatable join views, 5–12
UPDATE command
  foreign key references and, 7–15
  freeing space in data blocks, 3–9
update intensive environments, 10–11
update restrict, 7–15
updates, location transparency and, 21–7
user processes
  allocate PGA's, 9–26
  connections and, 9–30
  dedicated server processes and, 9–32
  sessions and, 9–30
  shared server processes and, 9–37
user program interface (UPI), 9–41
USER pseudo–column, 18–6
USER_ views, 8–5
USER_TAB_COLUMNS view, 13–18
USER_TABLES view, 13–18
USER_UPDATABLE_COLUMNS view, 5–12
users
  access
    procedure and package verification, 14–15
    rights, 17–2
  associated with schemas, 5–2
  auditing, 19–11
  authentication of, 17–3
  coordinating concurrent actions of, 1–30
  database links and, 21–5
  dedicated servers and, 9–32
  default tablespaces of, 17–6
  licensing by number of, 17–14

users (*continued*)
licensing of, 17–12
multi–user environments, 1–2, 9–4
names of, obtaining, 8–2
password encryption, 17–4
privileges of, 1–35
See also privileges
processes of, 1–20, 9–5
profiles of, 1–38, 17–11
PUBLIC user group, 17–7
resource limits of, 17–8
See also resource limits; security domains
restrictions on resource use of, 1–37 to 1–39
roles and, 18–10
for types of users, 18–11
schemas of, 1–33
security domains of, 1–35, 17–2, 18–12
single–user Oracle, 9–3
tablespace quotas of, 1–37, 17–6
tablespaces of, 1–37 to 1–39
temporary tablespaces of, 1–37, 17–6
user names, 1–35
See also schemas

V

V$LICENSE view, 17–13
V_$ and V$ objects, 8–7
VALID status, 16–3
See also name of object to which status applies
VARCHAR datatype, 6–3
VARCHAR2 datatype, 6–3
non–padded comparison semantics, 6–3
similarity to RAW datatype, 6–8
when to use, 6–3
variables, in stored procedures, 11–9
viewed tables. *See* views
views
ALL_UPDATABLE_COLUMNS, 5–12
altering base tables and, 16–4
auditing, 19–7, 19–8
constraints and triggers prohibited in, 5–9
constraints indirectly affect, 7–5
data dictionary use of, 8–3
DBA_UPDATABLE_COLUMNS, 5–12
definition expanded, 16–4
dependency status of, 16–4
histograms, 13–10
how stored, 5–9
indexes and, 5–11
INVALID status, 16–2
maximum number of columns in, 5–9
NLS parameters in, 5–11
object dependencies and, 5–12
overview of, 1–9 to 1–11, 5–8 to 5–16
partition views, 5–13
prerequisites for compilation of, 16–4
privileges for, 18–5
security applications of, 18–6
SQL functions in, 5–11
USER_UPDATABLE_COLUMNS, 5–12
uses of, 5–10
VALID status, 16–3
See also dependencies
virtual memory, 9–15
virtual tables. *See* views

W

waits for blocking transaction, 10–13
writers block readers, 10–13

Reader's Comment Form

Oracle7™ Server Concepts
Part No. A32534–1

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 topic, chapter, and page number below:

____
____
____
____
____
____
____
____

Please send your comments to:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.

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

____
____
____

Thank you for helping us improve our documentation.